Automating Component-Based System Assembly

A Thesis
Presented to
The Academic Faculty

by

Gayatri Subramanian

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

School of Electrical and Computer Engineering
Georgia Institute of Technology
August 2006

Automating Component-Based System Assembly

Approved by:

Dr Panagiotis Manolios, Committee Chair
College of Computing
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Douglas M Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: May 8, 2006

To Shyam and my family

ACKNOWLEDGEMENTS

I owe my most sincere and deepest gratitude to my research advisor Dr. Panagiotis (Pete) Manolios. The research work that I started with him, about a year back, has laid a strong foundation for my thesis. He has been a tremendous mentor to me, guiding the research right from the start till the very end. His suggestions, recommendations, and comments have been invaluable to me.

I would also like to thank Daron Vroon a student colleague, who has been a constant source of support and played a key role especially in implementing a part of the component-based system assembly tool.

I am extremely grateful to John Chilenski from Boeing Commercial Airplanes for funding this research project and providing us with industrial strength examples to test our component-based system assembly tool. I also wish to thank Dr. Sudhakar Yalamanchili and Dr. Douglas M Blough for their helpful suggestions while they were on my thesis reading committee.

I give a special thanks to all my other student colleagues for supporting and helping me along the way. Finally words alone cannot express the encouragement and support I received from Shyam, my family, and my room-mates right through till the completion of my thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Owing to advancements in component re-use technology, component-based software development and design (CBSD) has come a long way in developing complex commercial software systems while reducing software development time and cost. However, assembling distributed resource-constrained and safety-critical systems using current assembly techniques is a challenge. Within complex systems when there are numerous ways to assemble the components unless the software architecture clearly defines how the components should be composed, determining the correct assembly that satisfies the system assembly constraints is difficult. Component technologies like CORBA and .NET do a very good job of integrating components, but they do not automate component assembly; it is the system developer's responsibility to ensure that the components are assembled correctly.

In this thesis, we first define a component-based system assembly (CBSA) technique called "Constrained Component Assembly Technique" (CCAT), which is useful when the system has complex assembly constraints and the system architecture specifies component composition as assembly constraints. The technique poses the question: *Does there exist a way of assembling the components that satisfies all the connection, performance, reliability, and safety constraints of the system, while optimizing the objective constraint?* By answering this question, we either assemble the components appropriately or we determine that such an assembly does not exist.

To implement CCAT and demonstrate its applicability, we present a powerful framework called "CoBaSA". The CoBaSA framework includes an expressive language for declaratively describing component functional and extra-functional properties, component interfaces, system-level and component-level connection, performance, reliability, safety, and optimization constraints. To perform CBSA, we first write a program (in the CoBaSA language) describing the CBSA specifications and constraints, and then an interpreter translates the

CBSA program into a satisfiability and optimization problem. Solving the generated satisfiability and optimization problem is equivalent to answering the question posed by CCAT. If a satisfiable solution is found, we deduce that the system can be assembled without violating any constraints. Such satisfiability questions can be efficiently handled by taking advantage of advances in current Boolean satisfiability (SAT) methods.

Since CCAT and CoBaSA provide a mechanism for assembling systems that have complex assembly constraints, they can be utilized in several industries and domains, *e.g.*, avionics industry, automotive industry, package-configuration, and hardware electronics. We focus on the avionics industry, as avionic systems are a good example of large reliable distributed systems that have strict connection, performance, reliability, and safety guidelines expressed as assembly constraints. We demonstrate the merits of CoBaSAby assembling an actual avionic system that could be used on-board a Boeing aircraft. The empirical evaluation shows that our approach is promising and can scale to handle complex industrial problems.

CHAPTER I

INTRODUCTION

CBSD has simplified the task of software developers, by allowing them to re-use software components, when designing complex systems. At a high-level, developing a system from components can be relatively simple; the developers select the required components, prepare them for integration and then assemble the components into a system. The assembly is straight forward, when the software architecture specifies exactly how the components are to be put together, but it becomes difficult when there are numerous ways in which the components can be composed, and the software architecture expresses component composition as assembly constraints. Several industries who face this challenge have shifted focus from straight-forward CBSA techniques to CBSA techniques that tackle assembly constraints.

A good example is the avionics industry where software has become one of the major costs of developing aircraft. To curtail costs, the industry has moved away from federated systems—where subsystems use dedicated processing and I/O components—to integrated modular avionic (IMA) systems. IMA systems allow multiple subsystems to share the same resources and can lead to drastic savings in power, weight, development and maintenance costs, and overall efficiency [12, 48]. The avionics domain is an example domain where the number of components is large and it is tedious to assemble the components manually.

In this chapter, first we will consider the limitations of current assembly techniques within CBSD, and then we will look at our proposed CBSA technique to overcome those limitations. Then in Section 1.3, we will briefly describe the framework that implements our CBSA technique, and lastly we will give the overall outline of the thesis.

1.1 Limitations of Current Assembly Techniques

Components are primarily characterized by their functional and extra-functional properties. Functional properties succinctly express what the component is meant to do (*i.e.*,

functional behavior of the component). Extra-functional properties describe the component connection, performance, safety, and reliability requirements (these are more abstract requirements). The software architecture specifies how the components should be assembled such that all the components' functional and extra-functional properties are satisfied; the architecture can be either expressed explicitly or expressed as assembly constraints.

When there are a large number of components then the number of ways in which the components can be assembled is exponential, but only a subset of these assemblies will satisfy the component and system functional and extra-functional properties. When the software architecture explicitly describes how each component interface is to be composed with other components [45] then current assembly techniques do well. However, when there are several components that have similar interfaces and the architecture is not specific on how the components are composed, then current CBSA techniques are insufficient as they can not automatically search through the various possible assemblies and find a correct assembly that satisfies the assembly constraints. Current well-known CBSD standards like CORBA [42], and .NET [36], are very good at developing complex large-scale software systems, but they rely on assembly techniques that require the software architect to manually figure out how the components are to be composed.

CBSA techniques that tackle assembly constraints require the underlying component technology to place equal emphasis on both extra-functional properties and functional properties. However, most well-known technologies do not give enough importance to extra-functional properties when assembling the components [24]. For example, if there are hundreds of software applications to be allocated on a limited number of processing units, current assembly techniques cannot determine an optimal assembly that satisfies the extra-functional constraint that the processing units are load balanced. Extra-functional properties play a critical role in assembly of large reliable distributed systems. For instance, consider an avionic systems, where the number of components is very large and the components depend on one another for various extra-functional requirements (*e.g.*, resources like processing-time, memory, bandwidth, and power that are available in limited quantities on-board an aircraft). In such a system, manually determining how each component

obtains its required resources from the other components, in such a way that the system's performance, reliability and safety constraints are not violated is difficult.

1.2 CCAT: Constrained Component Assembly Technique

We define "Constrained Component Assembly Technique" (CCAT) as a CBSA technique within CBSD that overcomes the limitations of current assembly techniques by attempting to answer the following question.

Given a set of components and constraints; does there exist a way of assembling the components that satisfies all connection, performance, reliability, and safety constraints of the system, while optimizing the objective constraint?

If the answer to the above question is yes, then CCAT provides a solution for assembling the components such that all the constraints are satisfied. CCAT does not rely on the software architecture being explicitly specified, but assembles components based on the assembly constraints specified within the architecture. The following subsections look at various aspects of CCAT.

1.2.1 Components to be Assembled

In CCAT no distinction is made between commercial off-the-shelf (COTS) components, re-usable software components, and components developed for specific applications. All components are treated as black-boxes with well-defined interfaces, and functional and extra-functional properties.

CCAT assumes that the components have been developed and prepared for integration. To ease component assembly, it is best that software components are developed with the notion that they should be re-usable in many different systems and across different platforms and environments. In effect, this would mean a strong separation between the "interface" and the "implementation" of the component; making the component visible only through its interface, hiding the implementation details and as a result enhancing composability.

Experience indicates that interface and architecture mismatches do occur in unavoidable circumstances [23]. In such a situation, the component must be prepared for integration by

adding a suitable wrapper or glue-code that would mask any incompatibilities in programming languages, operating systems, and communication protocols.

1.2.2 Component Assembly Constraints

In CCAT the assembly constraints play an important role in determining which component assemblies satisfy the system functional and extra-functional properties and which assemblies do not satisfy those properties. Examples of constraints that affect component assemblies include connection, performance, reliability, and safety constraints. The constraints can be both system-level constraints (*e.g.*, overall system performance constraints) or component-level constraints (*e.g.*, component resource constraints). Through the following points, we explain how these constraints affect component assembly.

1. Connection constraints are constraints on the connections between component interfaces. These constraints determine how the components are put together. The connections are either physical connections between components, or logical connections between components that either depend on one another for a resource or communicate with one another. For example, if there are application and processor components within a system, then a constraint that says that 64-bit applications must be allocated only on 64-bit processors is a connection constraint. This connection constraint is different from exactly specifying which 64-bit processor the given 64-bit application must be allocated on. Connection constraints enable large amounts of flexibility in the way components are assembled.

2. Performance constraints can be divided into two categories namely resource constraints and temporal constraints. Resource constraints are constraints on extra-functional properties like memory consumption, power consumption, bandwidth availability and so on. Temporal constraints are constraints on extra-functional properties like worst-case execution time, periodicity, jitter, context-switching-time, and so on. For reliable systems, it is necessary that the component assembly satisfies the performance constraints.

3. The reliability of safety-critical systems depends on the redundancy within the systems. Hence reliability constraints can be thought of as redundancy constraints and separation constraints. Redundancy constraints express the required level of redundancy for critical components within the system to eliminate single-point failures. Separation constraints express the required separation between redundant components. For instance, a process and its redundant copy cannot be on the same server.

4. Safety constraints are general constraint on the system to ensure its safety. For example, a safety constraint could state that two components X and Y must have a physical distance of d meters between them.

5. In some cases it might be required that components be assembled such that an objective constraint is optimized.

1.2.3 Performing CBSA using CCAT

To perform CBSA using CCAT, first the components and constraints must be specified. The component specification must include the functional and extra-functional properties of the component and the component's interface description. Then the assembly constraints (*e.g.*, connection, performance, reliability, and safety constraints) must be specified. These constraints should incorporate constraints at both component and system level. Based on the component and constraint specifications, a satisfiability and optimization problem should be generated. If a satisfiable solution is found for the problem, then we can conclude that there is a way of assembling the components that satisfies all connection, performance, reliability, and safety constraints of the system, while optimizing the objective constraint if one is given.

Hence, CCAT allows us to check at design-time the composability of the components subject to constraints, without having to manually figure out how to physically put the components together to form a system, and test the resulting system at run-time to check for satisfiability of the performance, reliability, and safety constraints.

1.2.4 Automating CBSA

When the number of components and constraints is large, as is usually the case in large distributed systems, manually determining the assembly that satisfies the constraints is labor-intensive. Furthermore, as system design evolves and components are modified, the component specifications and constraints and system constraints often undergo major revisions, and determining whether or not the components can be composed and assembled in a way that satisfies the connection, performance, reliability, and safety constraints becomes more and more difficult and labor-intensive. Automating the CBSA process is important, so that it can be repeatedly used during the course of system design and evolution. This problem is of interest to companies such as Boeing and Airbus, who build airplanes by assembling components. In fact our interest in this problem arose from our interactions with Boeing.

1.3 CoBaSA

We have developed a general framework called "CoBaSA" (Component-Based System Assembly) that implements CCAT. Figure 1 illustrates CoBaSA being used to perform CBSA in the avionics domain. CoBaSA includes the following:

- An expressive language for declaratively describing component interfaces, component functional and extra-functional properties, and system-level and component-level constraints.
- An interpreter to convert the CBSA problem into a satisfiability and optimization problem.
- A pseudo-boolean solver to solve the constraints using SAT-based methods.
- Produces a component-based system assembly that satisfies all requirements and constraints.

CCAT is useful for assembly within any large generic component-based system. Hence, the CoBaSA system, which is based on CCAT, can be used to automatically solve any general case of CBSA, as long as the required component and constraint specifications are given.

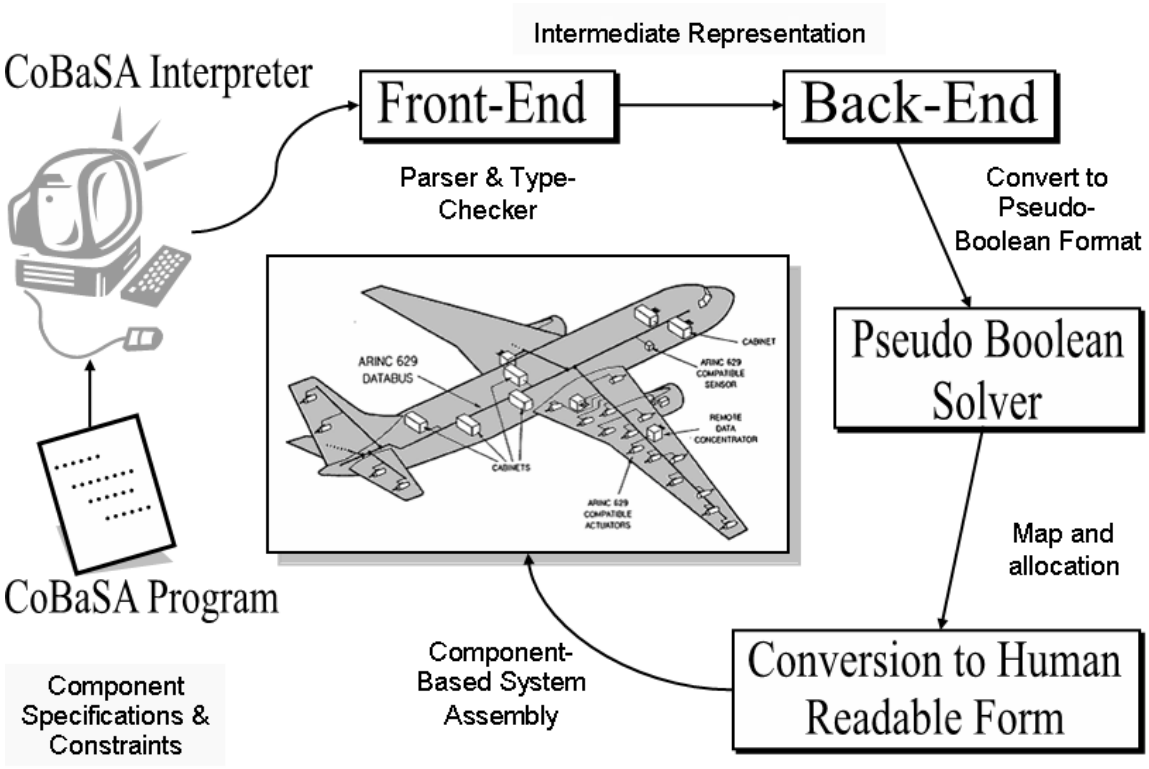

Figure 1: CoBaSA: Automating Component-Based System Assembly

The CoBaSA system is comprised of the CoBaSA language, the CoBaSA interpreter, and a Pseudo-Boolean solver. The CoBaSA language includes an object-oriented type system for describing components and embeds Common Lisp, giving users a powerful functional programming language. It also includes a constraint specification language that introduces the notion of a map, map constraints, and universally quantified statements involving arbitrary boolean expressions and summations. The CoBaSA language is designed to be general enough to allow most CBSA instances to be expressible by a CoBaSA program (*i.e.*, a program written in the CoBaSA language).

The CoBaSA interpreter consists of 2 parts: front-end and back-end. The front-end parses and type checks the program and generates an intermediate representation that is given to the back-end. The back-end simplifies and converts the objects, maps, constraints, quantified relational and arithmetic constraints, boolean formulas, *etc.*, in the intermediate representation to pseudo-boolean constraints and to CNF (Conjunctive Normal Form) clauses. The resulting constraints are then handled by the pseudo-boolean solver PBS [2], which takes advantage of current advances in SAT solving technology [70], to produce a component-based system assembly that satisfies all requirements and constraints.

1.4 Outline of the Thesis

The thesis consists of chapters on the following topics:

- Component-Based Software Development and Design
- Background and Related Work
- Pseudo-Boolean Solvers
- CoBaSA
- Integrated Modular Avionics
- Empirical Evaluation
- Future Work and Conclusion

A brief overview of CBSD concepts and methods is given in Chapter 2, and the limitations of current assembly techniques and the role of extra-functional properties in CBSA,

are emphasized. Chapter 3 presents the background and related work that has been done in CBSA, and motivates the development of CoBaSA. Since CoBaSA uses a pseudo-boolean solver to solve a CBSA instance, we briefly describe the salient features of pseudo-boolean solvers in Chapter 4. Next, the entire CoBaSA system along with language, interpreter, and pseudo-boolean solver is elaborately described in Chapter 5. Several examples are given to illustrate the descriptiveness of CoBaSA language and to explain the working of the CoBaSA system.

Since avionic systems cannot be assembled using current assembly techniques [40], in Chapter 6 we describe how CoBaSA can be used to model the assembly problems encountered in IMA systems. IMA systems are a good example of large safety-critical distributed systems that have strict connection, performance, reliability, and safety guidelines. We test the performance of CoBaSA, in Chapter 7, by assembling an actual avionic system that could be used on-board a Boeing aircraft. The empirical evaluation shows that our approach is promising and can scale to handle complex problems. Finally, we briefly discuss future work and conclude in Chapter 8.

CHAPTER II

COMPONENT-BASED SYSTEM DEVELOPMENT AND DESIGN

CBSD enables the construction of large complex software systems by integrating components. The potential benefits of CBSD include greater reliability, lower development costs, shorter development cycles, less testing and validation, more flexibility and reuse, etc. [25, 62].

In this chapter, first we will discuss basic definitions in CBSD that are relevant and then we will consider some of the generic obstacles in CBSD (in Section 2.2). This will be followed by a brief description of the functional and extra-functional properties of a component and the importance of the extra-functional properties for component assembly in Section 2.3. Lastly we will compare the 2 CBSD views, namely component-centric and system-centric (in Section 2.4).

2.1 Basic CBSD Definitions

The aspects of CBSD that we are interested in depend on the following definitions: component, component interface, software architecture, component assembly, and component extra-functional properties. The first four component definitions are discussed in the following subsections, but component extra-functional properties is discussed in Section 2.3.

2.1.1 Component

Many definitions for a component have been put forth, as it is difficult to zero-in on a single formal definition of a component that works for all scenarios. Below is a generic definition by Szyperski in [58].

Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system.

Specifying that software components are binary units, allows more black-box re-use [58], as the code becomes independent of the compiler, linker and environment under which it was developed. It is essential that the components are independently produced and acquired, to allow for interoperability of components from multiple vendors. In addition, if components are independently deployable, it would enable more robust integration with third party components.

Another definition by Crnkovic [14], which is similar to Szyperski's definition but easier to understand, states that *a component is a re-usable unit of deployment and composition that is accessed through an interface.* In CCAT, it is sufficient for us to assume that a component has the following properties.

- It is a binary unit that is re-usable.
- It has a precise well-defined interface.
- It is composable with third-party components.

As stated earlier, there are 3 types of components: COTS components, re-usable software components, custom-built (or application-specific) components. Obviously a system built out of purely custom-built components, gives the designer full control over the design, while tremendously increasing the cost in terms of time, money, and effort. COTS components on the other-hand, offer low costs in terms of time and money, but they place the burden on the designer, because they tend to be complex and unstable and provide little information about their complete behavior [14, 23, 62]. The usability of re-usable components depends on whether they were originally implemented with re-use in mind. The usability is higher when the components are developed with the notion that they will be re-used in future, as compared to the usability of custom-built components that need to be adapted to make them reusable. The designer has to weigh the pros-cons of various options, before selecting the set of components that he wants to use to build the system.

2.1.2 Component Interface

The interface of a component specifies the only access points to the component [14]. An interface must provide all the information needed to understand the operation of component

code, but keep the code inaccessible. In most CBSD techniques, the interface is defined by specifying the component's operations and context dependencies. For component assembly purpose, we will assume that the interface is defined by specifying what it provides and what it requires, or more formally as a set of input/output ports.

Since a component is accessible only through its interface, the interface must be precisely defined, to avoid any ambiguities. For instance, an interface description language (IDL) is used by CORBA [42] to specify the component interfaces. In CoBaSA, we use the CoBaSA language to specify component interfaces; the component interface specification is not separated from the specification of the remaining component properties as is usually done.

2.1.3 Software Architecture

A frequently cited definition for software architecture is the one given by Bass et al. [4]: *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*

In other words, the software architecture specifies the software components that make up the system and the topology or structure that describes how the components are composed. The architecture specifies the component relationships and the allowed component interactions. Additionally it specifies the "externally visible" properties (*e.g.*, performance characteristics, shared resource usage) of each component and how the properties of each component affects the properties of other components that are dependent on it. However, the definition is indifferent as to whether the architecture for a system is a good one or a bad one, meaning that the architecture may or may not meet the behavioral and performance requirements of the system.

2.1.4 Component Assembly

Cheesman et al. [11] define component assembly as *the process of pulling together components, and existing software assets into a working system.* In most component models, the developer connects all the required components, to produce a valid component-based system

that can be deployed [14]. It is the developers responsibility to ensure that the components are assembled according to the specification of the architecture.

Since the architecture specification does not guarantee the behavioral and performance requirements of the system, it implies that an assembly based on such an architecture will also not guarantee the behavioral and performance requirements of the system. To ensure that the component assembly meets the required behavioral and performance specifications, either we devise mechanisms to evaluate the architecture or we express the architecture structure as assembly constraints that would ensure that the requirements are met. Expressing the architecture as assembly constraints makes component assembly very flexible; thus making it easier to find the best possible assembly that satisfies the given behavioral, performance, reliability, and safety requirements of the systems. For instance, a constraint that states that 64-bit applications must be hosted on 64-bit processors gives more flexibility, than the architecture explicitly specifying for each 64-bit application which particular 64-bit processor it should be hosted on.

The separation of concerns between selecting and preparing components for integration, and assembling components is important. While assembling components, it is not necessary to consider the interface mismatches, architectural incompatibilities, and the glue code required for the components to work together. The components should be treated as black boxes and their assembly should be based solely on the assembly constraints.

2.2 Challenges in CBSD

In an ideal world any complex software system can be neatly partitioned and broken down into a number of components. For a system wherein all the components are available with compatible interfaces and all functional and extra-functional properties of the components are explicitly known, CBSD can be as easy as assembling Lego-blocks [14]. However, in reality CBSD is far more complex as indicated by the following points.

- In several large complex systems, assembly of components is not as intuitive as it may appear. The number of components is too large and the constraints on how they should be assembled is very complex. Current assembly techniques do not provide

ways to express the constraints and to find an assembly that satisfies the constraints.

- Most often the components' functional and extra-functional properties are not completely known or they are imprecisely defined. This is more so, for extra-functional than functional properties, because extra-functional properties tend to be abstract.
- It is difficult to develop components as reusable entities. There can be three reasons for this; either the component implementations are not general enough to be used over different platforms and architectures, or the interfaces are not general enough to be used over a broader range of systems, or there is insufficient separation between the component implementation and interface.
- Fixing interface and architecture mismatches, or incompatibilities by generating glue-code, can prove to be expensive and tedious [23].
- Inability to accurately predict system properties from known component properties is a major drawback. For instance in the general case, the reliability of the system cannot be predicted from the known reliabilities of the components. However, if the components are trust-worthy or have certifiable properties, then system properties can be predicted from certifiable properties of components [8, 64].

In spite of the many challenges, current trends in component-based approach [25] are successful in making the real world as close as possible to the ideal world of component-based development. A majority of the desktop and Internet applications that we use today take advantage of component-based technologies like CORBA [42], J2EE [57], and .NET [36]. These general-purpose component technologies work well when the hardware resources are abundant, but cannot cope with assembly of component systems that have strict connection, performance, reliability, and safety constraints [40, 14]. These additional requirements call for new techniques and new methods that go beyond functionality of components.

2.3 Extra-Functional Properties

As stated in Section 1.1, every component is characterized by a set of functional and extra-functional properties. Functional properties symbolize the functional behavior of the components and they form the basis on which components are selected. Extra-functional properties (also called "non-functional" specifications, or "ilities", a term coined by Mary Shaw at the Software Engineering Institute (SEI)), which are not the highest priorities in CBSD, are component and system characteristics that characterize overall behavior, but may not be expressible by functions. Examples of extra-functional properties include properties related to connection constraints (*e.g.*, a property that states that 64-bit applications should be allocated on a 64-bit processors), resource constraints (*e.g.*, computation power, memory consumption, bandwidth, *etc.*), temporal constraints (*e.g.*, execution time, latency, periodicity, deadlines, *etc.*), reliability, safety, robustness, and security [14].

While different component technologies have considered the functionalities of components, few have considered the specification and verification of extra-functional properties that leak through the component's interface and affect overall system behavior and quality [15]. Extra-functional properties play a critical role in assembling the components, co-ordinating concurrent access to shared resources (resource constraints), and establishing valid execution orders of component services [14] (temporal constraints). In addition they express the safety and reliability requirements of components and systems.

There are two difficulties in expressing the extra-functional properties. First, either only imprecise definitions of the properties are available or the properties are too abstract; second, relating the component properties to the system properties is challenging. As stated earlier in Section 2.2, deriving system extra-functional properties from component extra-functional properties is a difficult task. Another problem posed by extra-functional properties is that even if they are clearly expressed, it is difficult to statically test if the properties will hold in the system assembly at design-time rather than dynamically at run-time. The CoBaSA framework allows you to specify the extra-functional properties clearly and statically test that these properties hold when the components are assembled.

2.4 Component-Centric versus System-Centric

CBSD can be viewed from two angles, namely system-centric and component-centric [44]. Component-based system integration is used to compose components in the component-centric view. More emphasis is placed on the semantic issues of composing the components and hence, the middle-ware technologies are primarily concerned with standardizing external component properties, interfaces, packaging, and inter-component communication protocols. For example, CORBA is a component-centric, middle-ware architecture that provides a communication standard, which enables a CORBA-based program from any vendor, network, operating system, and programming language, to inter-operate with a CORBA-based program from the same or another vendor, network, operating system, and programming language [42]. Here the interface is built into the development of components and CORBA provides a platform for these components to be integrated. The emphasis is more on the functional characteristics and not so much on the extra-functional characteristics that affect the way the components are assembled.

A system-centric view of CBSD concentrates on developing systems as an assembly of communicating black-box components, analyzing resulting system properties, and generating "glue" code (when required) that binds system components. In the system-centric view before assembly is performed, the components must be developed and prepared for integration so that incompatibilities in programming languages, operating systems, and communication protocols between the components are fixed. It is easier to implement CCAT in the system-centric view than in the component-centric view. Since it based on the principle that components should be composed in a way that the component-level and system-level requirements and constraints are satisfied.

CHAPTER III

BACKGROUND AND RELATED WORK

CBSD is a well-established field of study and overviews describing CBSD trends and obstacles are available in several books [14, 25, 58, 62, 11]. Although CBSD predominantly uses pre-existing components, care must be taken when re-using components, as they may behave in unexpected ways in new environments. If proper testing and verifying techniques are not used, disastrous consequences are possible as occurred with the Ariane 5 [67]. While there has been work on checking configurations of systems developed using CBSD (*e.g.*, [5, 8]) and on verifying component assemblies [63], automated CBSA requires further attention [16, 13].

The pertinent background and related works are highlighted in this chapter. First in Section 3.1, the contributions of various component technologies toward development of component systems that satisfy performance and reliability constraints are described. In these component technologies the assembly techniques adopted are more or less straightforward. Next, the relevant work toward improving component specifications is discussed. Several people have recognized that constraint solving techniques are necessary to check for satisfiability of system constraints. Some of these constraint solving and optimization algorithms are highlighted in Section 3.4.

3.1 Component Technologies

Component technologies are classified into two groups; ones that are general-purpose component models for developing desktop and web applications like .Net [36], J2EE [57], CORBA [42], RT-CORBA [53], Minimum-CORBA [41], and CIAO [65] and ones that focus on the needs of safety-critical distributed systems like PECT [64], Koala [61], PECOS [68], and Autocomp. [52]. The difference between the two groups depends on the emphasis each of them places on extra-functional requirements and constraints. In standard component

technologies the performance, reliability, and safety constraints are enforced at run-time, and not at design time when the components are assembled. Component technologies for reliable systems require that the constraints be satisfied at design time [40]; the best effort allocation policy of standard component technologies is not sufficient for systems with complex run-time constraints.

However both groups do not go very far beyond straight-forward assembly techniques. The assembly is manually done by the developer based on a detailed architecture description [14]. It is up to the developer to ensure that the assembly satisfies the high-level system constraints.

3.2 Software Architectures

Software architecture expresses how each component is connected to other components, and it gives a detailed model of the component interactions [60, 45]. Such a specification is too restrictive, when there are several ways to assemble a given set of components.

Software architecture [47] has evolved to the point that effective tools for the development and analysis of software architectures are available [4, 54]. These include tools and techniques for automatically assembling components when given a detailed architecture description [27, 9]. While these tools can resolve the semantic issues of integrating components whose interfaces mismatch, they again do not generate the architecture themselves. For example, the GenVoca tool [7, 6] is a domain independent tool that generates a hierarchical software system from component specification and composition rules. The tool is effective as long as the software architecture specifies how each component is connected to other components. These tools have no way of dealing with architecture descriptions expressed as assembly constraints.

Automatic component deployment [30] and dynamic component redeployment [37] make adjustments or modifications to the component assembly (once it has been deployed) to satisfy the run-time constraints of the deployment architecture. This approach is good when sufficient run-time environment information is unavailable when the components are assembled.

3.3 Component Specification

CBSD faces many stumbling blocks; one of them is to develop specifications for components. There has been considerable work on this, including work on architecture description languages (ADL). After comparing various ADLs, Medvidovic et al. [35] make a point that existing ADLs lack in their support for expressing extra-functional properties that are essential to express assembly constraints. Hence, there is need to develop specifications languages with full support for expressing these properties.

Unified modeling language (UML) [50, 49] is the de-facto standard for modeling software applications. Even though it was originally developed for modeling object-oriented systems, Cheesman et al. [11] show how to successfully exploit UML for component specification. In addition UML includes an object constraint language (OCL) [43] that is textual language for defining component constraints as logical expressions. While UML is a good modeling language, most component technologies use their own specification language to have more control over language features that affect the integration and assembly of components. Moreover tools for automating system assembly for UML do not exist.

3.4 Constraint Solving

Components can have two types of extra-functional constraints; namely relational arithmetic constraints and Boolean constraints. Constraint solvers such as Pseudo-Boolean solvers allow representation of both boolean and relational arithmetic constraints.

Reliable component-based systems have been built using a genetic algorithm [22] that maps component services to real-time tasks on the operating system kernel. Tindell et al. [59] have considered allocating real-time tasks to processors in a distributed system using stimulated annealing. Both these methods, do not offer ways to express constraints other than real-time constraints (relational constraints) on the components. The constraints have to be hard-coded into the algorithms as there is no mechanism for expressing high-level constraints in the two methods.

Pure SAT-based techniques using quantified boolean formulas have been used for allocating tasks onto reconfigurable nodes within a network of connected FPGAs [21]. The

method described here doesn't consider the resource-constraints and real-time behavior of the tasks to be allocated.

CHAPTER IV

PSEUDO-BOOLEAN SOLVERS

CoBaSA implements CCAT by translating the given CBSA instance into a satisfiability and optimization problem and then solving the generated problem using a pseudo-boolean solver. A pseudo-boolean solver solves constraints on pseudo-boolean variables, which are variables that have values 0 or 1 in linear context and values `True` or `False` in boolean context. The constraints on pseudo-boolean variables can be both relational linear constraints and arbitrary boolean formulas in conjunctive normal form (CNF) [1, 2, 55, 56]. The linear pseudo-boolean constraints (PB-constraints) are of the form:

$$\sum_{i=1}^{n} a_i v_i \ R \ c$$

where, c and each a_i is a constant integer value, each v_i is a variable that ranges over the values $\{0, 1\}$, and R is either '$\leq$', '$\geq$', or '$=$'. Boolean formulas in CNF are represented as conjunction (logical and) of clauses, where each clause is a disjunction (logical or) of literals and each literal is either a boolean variable or its negation. For instance,

$$(X_1 \vee \neg X_2 \vee X_4) \ \wedge \ (\neg X_1 \vee X_3)$$

is a boolean formula in CNF having two clauses over boolean variables $X_1 ... X_4$.

There are 3 main techniques for solving satisfiability and optimization problems on pseudo-boolean variables.

1. Pseudo-boolean problems can be expressed as 0-1 integer programming problems and solved using generic integer linear programming (ILP) techniques.
2. Pseudo-boolean problems can be translated into CNF clauses and solved using generic SAT solvers, wherein the solvers are not modified to handle PB-constraints.
3. Specialized techniques that solve pseudo-boolean problems without converting or translating any of the constraints. These are pure pseudo-boolean solvers catering

only to pseudo-boolean problems.

In the following subsections we describe the essentials of these techniques, without going into too much detail. The emphasis is on the algorithms used in the techniques. In Section 4.1, we describe 3 integer programming algorithms that are most used. Next in Section 4.2, we start out with the basics of SAT solvers and then go on to explain how they can be used to solve pseudo-boolean problems. Finally in Section 4.3 we describe the specialized techniques that cater to only pseudo-boolean problems. Since most of the pure pseudo-boolean solvers are based on either modified SAT techniques or modified ILP techniques, we describe ILP and SAT techniques in this chapter even if those techniques are not directly used within CoBaSA. While describing the techniques, we also describe some pseudo-boolean solvers.

4.1 0-1 Integer Linear Programming Techniques

Computationally effective algorithms (polynomial-time algorithm [29]) for solving linear programming problems are very widely used within the operations research and artificial intelligence community. Similar computationally effective techniques do not exist for ILP problems (ILP is NP-complete [46]), as ILP is computationally more complex than its linear programming counterpart. Associated with every integer program is a linear relaxation program formed by dropping the integrality restrictions. Most ILP algorithms solve an ILP problem, by solving its linear relaxation.

0-1 ILP is a special case of ILP with the additional constraint that every variable $x \in \{0, 1\}$. A pseudo-boolean problem can be converted into a 0-1 ILP problem, by converting the CNF formulas into linear constraints and by retaining the PB-constraints as such. 0-1 ILP problems are tackled by the same algorithms that tackle general ILP problems. The commonly used ILP techniques can be succinctly described as follows.

1. In branch-and-bound [31] algorithms, a linear relaxation of the 0-1 ILP problem is solved, based on which two new subproblems are created by branching on a variable that has a fractional value in the solution.

2. The fundamental idea behind cutting planes [39] techniques is to keep adding constraints to the linear relaxation of the program, until the optimal solution takes on integer values.

3. Branch-and-cut [38] algorithms reap the benefits of both branch-and-bound algorithms and cutting-plane techniques, by reducing the total number of linear relaxations that have to be solved.

CPLEX [26] is a commercially available solver that uses branch-and-bound and branch-and-cut techniques to solve 0-1 ILP problems (and even general ILP problems). Inspite of being highly optimized CPLEX does not take advantage of the boolean nature of the variables when solving 0-1 ILP problems.

4.2 SAT Solving

A boolean satisfiability (SAT) problem is usually represented as a propositional formula in CNF (SAT solvers cannot directly handle PB-constraints). The goal of a SAT solver is to find a variable assignment that makes the given propositional formula true. Recent advances in boolean satisfiability methods have lead to development of very efficient SAT solvers that can often solve SAT instances generated from industrial applications with tens of thousands or even millions of variables. The well known SAT solvers include MiniSAT [19] and zChaff [69].

The basic SAT algorithm lays the foundation for pseudo-boolean solvers that are SAT-based. In the following subsection we describe the basic algorithm adopted by current SAT solvers. Then in Section 4.2.2, we highlight how SAT solvers deal with PB constraints.

4.2.1 DPLL Algorithm

The latest state-of-the-art SAT solvers are all fundamentally based on the algorithm originally proposed by Davis, Putnam, Logemann, and Loveland (DPLL) [17]. Zhang and Malik in [70], have laid down the essentials of the DPLL algorithm, and have described the features used by the latest solvers that significantly improve the performance of the DPLL-style decision procedures.

For the propositional formula to be true, every clause in the formula must be individually satisfied, which implies that each clause must have at-least one true literal. If a variable assignment results in a conflicting clause, which is a clause that has all false literals, then that variable assignment will not be able to satisfy the formula. If there exist a clause that is a conflicting clause for all possible variable assignments, then it implies that the given propositional formula is unsatisfiable.

At the start of the DPLL algorithm, all the variables in the CNF formula are unassigned. An unassigned variable called a decision variable is selected, and the algorithm branches on the selected decision variable by assigning it a value. The formula is then simplified based on this decision and further reasoning is done to determine what variable assignments are further needed for the formula to be satisfiable given the current set of decisions variables. If all clauses are satisfied with the given set of decision variables, then a satisfying variable assignment has been found. If there exists a conflicting clause, then the DPLL algorithm backtracks and undoes the decisions that led to the conflict. While backtracking, new clauses are learned from the clauses that caused the conflict, this is called conflict-driven learning. If the algorithm backtracks back until the very first decision variable, then it implies that the SAT instance is unsatisfiable. If it is neither the case that all clauses are satisfied, nor is there a conflicting clause, then the algorithm branches again by selecting a new unassigned variable as the decision variable. The DPLL algorithm continues until either a satisfying variable assignment is found, or it has proved unsatisfiability.

Majority of current SAT solvers follow the basic DPLL algorithm, but they differ in their branching heuristics, deduction mechanisms, conflict-analysis, and learning mechanisms [70]. Variable state independent decaying sum (VSIDS) is widely used as an effective branching heuristic to select the next unassigned variable to branch on. Almost all modern SAT solvers incorporate boolean constraint propagation (BCP) in their deduction mechanism. BCP states that if for a certain clause, if all but one of its literals has been assigned false, then the remaining (unassigned) literal must be assigned true for this clause to be satisfied, which is essential for the formula to be satisfied.

4.2.2 SAT Solvers in Pseudo-Boolean Problems

SAT solvers are designed to handle CNF formulas and they cannot tackle linear PB-constraints. Hence, to solve a pseudo-boolean problem using the DPLL algorithm, the problem must first be represented as a CNF formula. Transforming pseudo-boolean constraints into CNF clauses can lead to an exponential blow-up, as a single PB-constraint can correspond to an exponential number of CNF clauses [1].

SAT solvers like MiniSAT+ [20], which cleverly translate PB-constraints to CNF clauses, perform better on certain pseudo-boolean benchmarks, in comparison to algorithms specialized for pseudo-boolean problems [33]. MiniSAT+ is a pseudo-boolean solver that converts the pseudo-boolean problem into CNF clauses and uses MiniSAT (a SAT solver) to solve the resulting SAT instance. In MiniSAT+, first the PB-constraints are normalized, then each constraint is translated into an arithmetic circuit, and then each circuit is translated into CNF clauses. Note that the translation does not lead to an exponential blow-up, as it is compact, and as much as possible implications between the literals in PB-constraint are preserved. Except for a few constraints, if nature of remaining constraints are more SAT-like, then it is good to convert the problem into an SAT instance and solve it using a SAT solver.

4.3 Pure Pseudo-Boolean Solvers

Pure pseudo-boolean solvers are in fact, either generalizations of SAT solvers adapted to deal directly with PB-constraints, or SAT solvers integrated with ILP techniques. SAT-based pure pseudo-boolean solvers use techniques such as boolean constraint propagation, conflict-analysis, and conflict-driven learning, by adapting these DPLL techniques to PB-constraints. Since these solvers use SAT-based techniques, their algorithms can be integrated into the DPLL framework, with a few tweaks and adjustments. Moreover, it is also possible to integrate ILP techniques such as branch-and-bound and cutting planes, into the DPLL framework. Note that the approach of converting pseudo-boolean formulations to propositional clauses seems to be competitive with pure pseudo-boolean solvers.

In this section we are going to consider the algorithms followed by 3 pure pseudo-boolean

solvers. These are not the latest or the most advanced solvers, but each of them follows a different approach algorithmically, which is of interest to us.

4.3.1 PBS

PBS [2, 1] is a backtrack-search pseudo-boolean solver and optimizer. PBS handles CNF clauses in much the same way as DPLL procedures. It uses VSIDS in its branching heuristics, it uses the 2-literal watching BCP mechanism to deduce the implied variable assignments for the CNF clauses, and it supports random restarts and non-chronological backtracking (refer to [70] for more information on VSIDS, 2-literal watching, random restarts, and non-chronological backtracking).

The BCP and conflict-driven learning mechanisms for PB-constraints are different from those for CNF clauses. For example when BCP is applied to a <= PB-constraint, a literal is implied to false, if its coefficient is greater than the goal minus the value of left-hand-side of the constraint computed based on the current variable assignment. If a <= constraint is a conflicting-constraint, then PBS learns a CNF clause that consists of true literals in the constraint whose sum of coefficients is greater than the goal.

Optimization is done in PBS iteratively by adjusting the goal (one step at a time) of the optimization constraint, until a satisfiable solution can no longer be found. The mechanism is inefficient because if the optimum maximum value of the objective function is 1000 and the solver has found an initial satisfiable solution that gives a value of 100 to the objective function, then the solver will have to iterate over 900 steps to reach the maximum objective value in the worst case.

4.3.2 Galena

Galena [10] is pseudo-boolean solver that combines DPLL-style decision procedures with cutting-plane techniques described in Section 4.1. Even though its performance is slightly erratic [33], we mention it here to demonstrate how cutting plane techniques can be used with PB-constraint solving.

Similar to PBS, Galena handles CNF clauses in a DPLL-like style. However, the PB-constraints are handled slightly differently. Galena employs a BCP procedure, wherein a

literal in a PB-constraint becomes implied as soon as it becomes necessary that its coefficient is required to satisfy the constraint. In addition to learning CNF clauses, galena unlike PBS can also learn PB-constraints. A learned PB-constraint has the potential of pruning more of the search space than a CNF clause. However, the steep overhead of manipulating PB-constraints can more than offset their pruning benefits.

The cutting plane technique computes a linear combination of a pair of PB-constraints and this mechanism is used to learn PB-constraints during the conflict-analysis phase. The steps include step-by-step elimination of implied variables, until a constraint that becomes unit after erasing the last decision assignment is obtained. In Galena, a cut is added as PB-constraints, only if the cut does not weaken the conflict and imply a wrong variable assignment.

Pueblo [55] is a another pseudo-boolean solver that employs the cutting plane technique. It is a combination of PBS (explained in Section 4.3.1) and cutting plane techniques for learning PB-constraints. Pueblo performs [33] better than either PBS or galena.

4.3.3 OPBDP

OPBDP (optimized pseudo-boolean Davis-Putnam enumeration) [3] is an implicit enumeration algorithm for solving pseudo-boolean optimization problems. OPBDP follows an approach similar to the branch-and-bound (described in Section 4.1) algorithm used in ILP.

Similar to how branch-and-bound algorithms solve linear relaxations of ILP problems, OPBDP solves weak discrete relaxations called pseudo-boolean unit relaxations for pseudo-boolean problems. Barth [3] has generalized the Davis-Putnam enumeration (DP) algorithm [18] (the DP algorithm was prior to the DPLL algorithm and it was resolution based) for the pseudo-boolean case by generalizing the unit clause resolution to pseudo-boolean unit resolution. In OPBDP, the solver performs pseudo-boolean unit resolution and branches based on the result of the resolution. By comparing the performance of OPBDP with CPLEX [26], Barth [3] concludes that branching methods based on discrete relaxations compare well with branching methods based on linear relaxations.

CHAPTER V

COBASA SYSTEM

CoBaSA is a tool that performs CBSA by implementing CCAT. It is a framework that includes an expressive component specification language, an interpreter, and a pseudo-boolean constraint solver PBS [2, 1]. To perform CBSA using CoBaSA, first the CBSA instance is written as a CoBaSA program, next the interpreter parses and type-checks the program and converts it into a suitable format for the pseudo-boolean solver, and finally the solver comes back with an answer and if the answer is a satisfiable solution, then the solution describes how the components are to be assembled.

In the subsequent 3 sections we will describe the features of CoBaSA, in the first section we will concentrate on the language, in the next section we will focus on the interpreter, and in the last section we will highlight how PBS works in CoBaSA.

5.1 CoBaSA Language

In this section, we give a brief overview of the CoBaSA language. As stated earlier the language is declarative, but allows you to embed Common Lisp code when functional programming capabilities are required. We begin with an informal tour of the language using examples to illustrate the syntax and semantics. The examples are simple and demonstrate how the CoBaSA language supports CCAT in generic component-based systems that have assembly constraints. Also note that the complete syntax and semantics of the language is given in Appendix A.

5.1.1 Type and Variable Declarations

5.1.1.1 Type Declaration

Our language supports basic data-types like boolean, integer (including bounded integer types called range), and string. In addition, it supports user-defined data-types called entities, which resemble Java classes without methods. Additionally, all data-types can be

organized into arrays (including multi-dimensional arrays). A component specification can be expressed through an entity definition, and instances of the component can be created by defining objects for the entity definition. The syntax of an entity definition is illustrated by the following example.

```
entity application {
  ;name string
  ;memory int
  ;processing-time int
  ;port-number int[4]=[25,53,80,110]
}
entity web-application extends application {
  ;port-number 80
  ;network-bandwidth int
  ;invoke-appl web-application
}
```

The first specification defines an application component by declaring an entity of type `application`. The fields within the entity definition represent the interface and extra-functional properties of the component. In the above example, an `application` is characterized by 4 fields: a name field of type string, two integer fields that express the application's memory and processing-time requirements (component extra-functional properties), and an integer array field that expresses the port numbers on which a application can send requests to (component interface). Note that every field in an entity definition begins with a ';' and specifies a field name and either a type or value for the field. If a value is given for a field, then that value holds for all instances of that component, *i.e.*, all application instances will have the same value for the `port-number` field. Since this is a declarative language a value can be set only once.

The second definition creates a `web-application` component which extends the `application` component. This means that a web-application entity inherits all the fields contained in

the application component, and it additionally defines new fields to characterize itself. A `web-application` is an `application` component that sends requests on `port-number` 80, has `network-bandwidth` as an additional extra-functional property, and has a field for specifying the `web-application` that it can invoke. Note that entities can be recursive, as is the case with the `web-application` entity, which contains a field that is of type `web-application`. When the value of a field is an instance of another component, then in fact the value is a pointer to that instance of the component. In this example, we will refer to `application` as the base entity and `web-application` as the extended entity. When both the entity and the extended entity have a field with the same name, type casting is used to resolve ambiguities, *i.e.*, `port-number` refers to the field within the `web-application` entity and `(application)port-number` refers to the field within the `application` entity.

5.1.1.2 Variable Declaration

After the components are specified, instances of components can be created by using variable declarations. Declaring a variable can be thought of as, either defining an object for an entity type, or defining a basic data-type variable. The following example illustrates the syntax of a variable declaration.

```
var ;int num = 5 ;string bar
var ;bool[2] x = [true, false]
var ;application A1 = {;"APPL_0001" ;256 ;1200 ; }
var ;web-application W1 = {; ;1000 ; ;"HTTP_APPL_0001" ;128 ;1200  ; }
```

The first line defines an integer called `num` with value `5` and a string called `bar` with no value specified. The second line creates an array of type `bool` with values `true` and `false`. The third line creates an object `A1`, which is an instance of the `application` component. Each ';' in the definition represents a field of the entity. The order of the fields in the object definition correspond to their order in the entity definition. Hence, the first field corresponds to the `name` field, the second field corresponds to the `memory` field, and so on. If a field has been given a value within the entity definition, then the value of that field can be left blank within the variable declaration. If no text appears between two semicolons (or between a

';' and a '}'), then it is interpreted as a blank field (*e.g.*, the `port-number` field is left blank within both `A1` and `W1`). A blank field can also be assigned a value at a later point. The fourth line creates a `web-application` object called `W1`. The second field of `W1`, specifies its `network-bandwidth` requirement. The third field, namely the `invoke-appl` field, is left blank. In `W1`, once the `web-application` fields have been specified, the definition fills the fields of the immediate ancestor type (namely `application`) and so on until all the ancestors are exhausted. Thus, in our example the fourth field corresponds to the `name` field of the `application` entity, the fifth corresponds to the `memory` field of the `application` entity, and so on.

To assign values to variables or fields of objects that have been declared but not assigned a value, our language provides an `assign` statement. The following code assigns values to `bar` and the `invoke-appl` field of `W1`.

```
assign bar = "hello world"
assign W1.invoke-appl = {; ;1500 ; ;"DNS_APPL_00120" ;64 ;1600  ; }
```

Note that a '.' is used between the object name and the field name, to access the value of that field within the object.

5.1.2 Maps and Field Constraints

5.1.2.1 Map Constraint

Once the designer declares types (components) and variables (instances of components), she can begin defining the system and component constraints. The central concept of the constraints section is that of *maps*. These are functions whose domain and range are both sets of variables. They can be thought of as mappings from resource consumers to resource providers. Rather than being defined, maps are constrained, and the CoBaSA system attempts to find a satisfying definition. In other words map constraints are connection constraints that specify how components should be mapped to one another. Figure 2 illustrates a simple component assembly example in a distributed computing network that involves `application` and `processor` components. The assembly of `processor` components with `application` components can be considered as a task of mapping `application`

```
entity processor {
  ;mem int
  ;proc-cycle int
  ;bit-64 bool
}
entity dsp-proc extends processor {
  ;appl-type string[4]=["signal","image","audio","video"]
}
entity network-proc extends processor {
  ;appl-type string[3]=["packet","routing","encription"]
}
entity application {
  ;appl-type string
  ;bit-64 bool
  ;mem-req int
  ;proc-cycle-req int
}

var dsp-proc[20] dsp =
  [ {; ;1024 ;25000 ;false},
    ...
    {; ;512 ;16000 ;true} ]
var network-proc[20] nw =
  [ {; ;1024 ;25000 ;true},
    ...
    {; ;2048 ;25000 ;false} ]
var application[1000] appl =
  [ {;"routing" ;true ;32 ;100},
    ...
    {;"audio" ;false ;128 ;400} ]

map appl-proc appl (dsp, nw)

constraint appl-proc ((mem-req, proc-cycle-req))
                     ((mem, proc-cycle) (mem, proc-cycle))
```

Figure 2: A Simple Component Assembly Example in CoBaSA

components to `processor` components.

Figure 2 first describes the entity and object definitions for both the components. `Processor` components have extra-functional properties like `mem`, and `proc-cycle`, and they may or may not support 64-bit (`bit-64`) applications. The `processor` entity is extended by 2 entities, namely `dsp-proc` and `network-proc`, each of which have a string array field to express the types of applications they can support. The last entity definition in the figure is for application components; the definition includes one string field to specify the type of the application, one boolean field to specify if the application is a 64-bit application,

and 2 integer fields to express minimum memory and processing-cycle requirements. In the given assembly example, there are 20 DSP processors (called `dsp`), 20 network processors (called `nw`), and 1000 applications (called `appl`) and we need to figure out which processor each application is mapped to by considering the various possibilities and finding one that satisfies the constraints.

In the simple assembly example, we wish to map each of the applications to one of the processors (either DSP or network) such that all the applications' extra-functional requirements are satisfied. The mapping constraint that maps applications to processors is given toward the end in Figure 2. A mapping constraint has the following syntax.

```
map <Map_Name> <Consumer_Components> <Provider_Components>
```

In general, a map command takes a name, followed by the consumer components, followed by the resource-provider components. The consumers and providers may each be a variable, an array of variables, or a list of variables and arrays of variables. A mapping has a built-in constraint: each instance of a consumer component is mapped to exactly one instance of a provider component.

5.1.2.2 Field Constraint

Additional constraints can be denoted in several ways. The first is by specifying field constraints *i.e.*, constraints on the extra-functional properties of the components involved in the map. If a component has performance constraints, then they can be expressed using field constraints. The field constraint tells the constraint solver which fields of the consumers draw from which fields of the providers. In our case, the memory and processing-cycle requirements of the applications will be provided by the `mem` and `proc-cycle` fields of both the DSP and network processors. The last constraint in Figure 2 represents the corresponding field constraint for mapping applications to processors. In our language, field constraints have the following syntax.

```
constraint <Map_Name> <Consumer_Field_lists> <Provider_Field_lists>
```

Each constraint statement is given a map name, and two lists. The first list has length equal to the length of consumer list of the map, and the second has length equal to the length of provider list of the map. The i^{th} member of the first list should be a list of fields of the corresponding item of the consumer list of the map, and similarly for the second list and the provider list of the map. All of these lists need to be of the same length. The consumer field lists correspond to the "needs" of the consumer, and the provider field lists correspond to the "resources" that the providers provide to meet these needs. The implicit constraint for these statements is that, for each provider there must be enough of every resource to meet the needs of the consumers that are mapped to it. In the simple assembly example, this amounts to saying that every processor must have at least as much memory and processing-cycles as all the applications running on it require.

5.1.3 Relational and Boolean Constraints

To express reliability and safety constraints, more expressiveness is needed than that offered by the map and field constraints. CoBaSA allows the designer to specify explicit constraints in the form of arbitrary expressions over boolean variables and what we call map references, and relational expressions over pseudo-boolean variables. The language provides `for_all` and `sum` constructs, for easily applying constraints to entire arrays. For example, we may want to say that 64-bit application must be mapped only to processor that has support for 64-bit applications. This can be accomplished by the following statements.

```
For_all a in appl {
  For_all d in dsp {
      ((appl-proc(a,d) and a.bit-64) implies d.bit-64)}
  and
  For_all n in nw {
      ((appl-proc(a,n) and a.bit-64) implies n.bit-64)}
  }
```

In this example, `appl-proc(a,d)` is a map-reference and is `True` if `appl-proc` maps application `a` to DSP processor `d`. The `For_all` statement applies the body to each of the

values in the array. In addition, `For_all` statements can range over a natural number, meaning they range from zero to one less than the number. For more complicated computations, the user can give arbitrary code in the Common Lisp programming language, which must return a value of the appropriate type. For boolean expressions, the result is caste to `False` if the value is `nil`, and `True` otherwise. For example, we can use this feature to say that an application should be mapped to a DSP processor only if the type of the application is supported by the processor. In the code shown below, the Lisp code is written within a let expression that returns true if `'at'` the type of application `'a'`, is within `'pt'` the array of types supported by DSP processor `'d'`.

```
For_all a in appl {
  For_all d in dsp {
    appl-proc(a,d) implies
    (let ((at a.appl-type)
          (pt d.appl-type))
       _(find at pt :test #'equal)_)}
  }
```

Another way to express CCAT constraints is through relation constraints. These involve arithmetic expressions over booleans and integers, where all the integer values must be known at compile time. In this context, boolean values are viewed as 1 or 0 rather than `True` or `False`. As with boolean expressions, Lisp code may be used for arbitrary computation. However, in this context the Lisp code must return an integer. A simple example of a relational constraint is the following, which limits the number of applications mapped to any network processor to 30.

```
For_all n in nw {
  Sum a in appl appl-proc(a,n) <= 30 }
```

```
entity server { ;bandwidth int}

entity processor {
 ;nw-bw int
 ;bw-req int
 ;percent-overhead [0..100]
}

entity application {
 ;proc-time-req int
 ;bw-req int
}

var server[10] sv = [ { ;2048} ... ]
var processors[100] pr = [ { ;512 ; ;12} ... ]
var application[1000] appl = [ { ;32} ... ]

map proc-sv pr sv
constraint proc-sv bw-req bandwidth
map appl-proc appl pr
constraint appl-proc bw-req nw-bw
```

Figure 3: An Example of Interdependent Maps in CoBaSA.

5.1.4 Interdependent Maps

In general, the semantics of CoBaSA require that only unassigned boolean variables be present in the satisfiability and optimization problem that is generated by the interpreter. However, there is a special form of relational constraint that can be used to set an unassigned integer value to some expression over boolean variables. This is useful when there are interdependent maps that have unassigned integer variables in them. Let us go back to the distributed computing network example given in Section 5.1.2, and add server components to the example. An application while requesting services from a server, requires some amount of network bandwidth from the server. The application makes its request through the processor on which it is hosted. This scenario looks like the code in Figure 3.

The value of the `bw-req` field within processor components is unassigned and depends on the bandwidth requirements of the applications that get mapped to it. This means that the bandwidth requirement of a processor from a server depends on the bandwidth requirements of the application. In addition, the processor incurs an overhead when negotiating services between application and a server. We can represent this using the special form of the

relational constraint as follows.

```
For_all p in pr {
  p.bw-req
  =
  Sum a in appl (* appl-proc(a,p)
                   (let ((bwr a.bw-req)
                         (po  p.percent-overhead))
                      _(ceiling (* (+ 1 (/ po 100)) bwr))_))}
```

This sets the bandwidth requirement of the processor to be the sum of bandwidth requirements of applications mapping to it increased by the percent overhead specified in the processor definition. The solver can then simultaneously solve the two mappings providing an assembly that will provide the necessary bandwidth to the processors and the applications.

5.1.5 Optimization

A CBSA instance can have number of possible solutions. It might be desirable to obtain a solution that maximizes or minimizes a given objective constraint. The objective constraint can be expressed as an arithmetic relational expression. For example in the distributed computing network example, it is desirable to map applications to processors, in a way that would minimize the overall bandwidth overhead. We can do this with the following command.

```
Minimize Sum a in appl Sum p in pr
           (* appl-proc(a,p) (let ((bwr a.bw-req)
                                  (po  p.percent-overhead))
                               _(* (/ po 100) bwr))_)
         <= 50
```

The above command says that the overall sum of the bandwidth overhead must be minimized, and the minimum value should be less than or equal to 50. In CoBaSA it is necessary

to provide an initial bound on the goal that is to be maximized/minimized.

Another common situation when an optimization constraint is useful is when the distances between components that map to one another has to be minimized to reduce wiring costs. For instance, it might be desirable to map the processors to servers that are closest to them in terms of network distances. If `x-pos` and `y-pos` denote the fields that express the x and y axis coordinates of the processor and server components. Then the objective constraint can be expressed as shown below.

```
Minimize Sum p in pr Sum s in sv
          (* proc-sv(p,s) (let ((px p.x-pos)
                               (py p.y-pos)
                               (sx s.x-pos)
                               (sy s.y-pos))
                           _(ceiling (sqrt (+ (expt (- sx px) 2)
                                              (expt (- sy py) 2))))_))
         <= 20
```

5.2 CoBaSA Interpreter

The CoBaSA system ultimately converts a CBSA instance into a satisfiability and optimization problem. The CoBaSA interpreter is responsible for the conversion, and the conversion is done using 2 parts namely the front-end and the back-end. The ensuing two subsections describe the functions of the front-end and the back-end.

5.2.1 Front-End

When a CoBaSA program is given to the CoBaSA system, it is in fact given to the front-end. The front-end reads the CoBaSA program and starts to parse and type check the program line by line. During the process of parsing and type-checking, if the front-end finds either a syntactic or semantic error, then it prints out the corresponding error message and stops the execution of the program. The parsing and type-checking is done according to the syntactic and semantic rules of the language. Appendix A describes the semantic errors

that are caught while type-checking.

Being a declarative language, a value can only be set once in CoBaSA, and so the type checker verifies that values are never set twice, and propagates defined values to all the variables. While parsing the program, the front-end evaluates the in-line Lisp expressions and expands `For_all` and `Sum` statements. When evaluating in-lisp expressions, the type-checker must ensure that the resulting value is boolean when the lisp expression occurs as a part of a boolean expression, and that the resulting value is an integer when the lisp expression occurs as a part of an arithmetic expression.

At the moment CoBaSA is designed to work only with a pseudo-boolean solver. Due to this, only constraints that allow conversion to pseudo-boolean format are currently allowed by the CoBaSA type-checker. Such limitations can be overcome, if the CoBaSA system is made more general by allowing the use of different types of solvers within the existing framework. For instance, if a problem had unassigned integers whose values had to be found by the solver, then an ILP solver (described in Section 4.1) would work best in that situation.

While the program is parsed and type-checked, the constraints are collected into an intermediate representation consisting of data structures, which are used later by the back-end to generate the satisfiability problem. The front-end uses 3 data structures, one for collecting map-constraints, one for collecting field constraints, and one for collecting explicit constraints (boolean and relational constraints). For each new map that the front-end comes across while parsing, it creates an entry for that map in the map data structure; the entry includes information about the map name, the set of consumer components, and the set of resource components. The data structure for storing field constraints is a hash table. The keys for the table are pairs $\langle P, F \rangle$, where P is a provider component and F is the name of a field of that component, and the values of the hash table are sets of triples, $\langle M, C', F' \rangle$, where M is a map, C' is a consumer component, and F' is the name of a field of C' such that if M maps C' to P, the field F' of C' draws from the field F of P. Before making an entry into the field constraints data structure, the type checker verifies that $P.F$ (the value of field F of object P) is an integer that has been assigned a value, and each $C'.F'$

is of type integer and has either been assigned a value or has been constrained to be equal to some arithmetic expression over map references and boolean variables (as shown in the example in Section 5.1.4).

5.2.2 Back-End

CoBaSA uses PBS as its constraint solver and since PBS can handle both PB-constraints and CNF clauses (explained in Section 4.3), the back-end converts map, field, and relational constraints into linear PB-constraints, and it converts boolean expression constraints into CNF clauses.

If M is the map name, C is the set of consumer component instances, and P is the set of provider component instances, then for each $c \in C$ the back-end generates a linear constraint of the form:

$$\sum_{p \in P} M(c, p) = 1.$$

where, $M(c, p)$ is the boolean variable (or map reference) that is true if the consumer c gets mapped to provider p. Since $M(c, p)$ can only take 0-1 values, the above constraint implies that $M(c, p)$ is true (*i.e.*, has value 1) only for one provider $p \in P$. In other words the consumer c must get mapped to only one provider from the providers in set P. Since a similar constraint is present for each $c \in C$, the total number of boolean variables that are generated for map M are $|C| * |P|$. By solving the above constraints one can tell which consumer component instance is mapped to which provider component instance.

The PB-constraint implied by a field constraint is that the provider component must meet the needs of all the consumers that map to it, under all maps. In other words, the "needs" of all the consumers that map to a given provider – by any mapping – need to be met by the appropriate fields of the provider. For an entry in the field constraint hash table with key $\langle P, F \rangle$, and value the set $\{\langle M^1, C_1, F_1 \rangle, \langle M^2, C_2, F_2 \rangle, \ldots \langle M^n, C_n, F_n \rangle\}$, back-end creates a PB-constraint of the following form:

$$\sum_{i=1}^{n} (C_i.F_i) M^i(C_i, P) \leq P.F$$

This constraints implies that the sum of requirements of fields of n consumer components must be met by the field F of provider component P. However, the constraint might

not be in the linear form expected by a pseudo-boolean solver (the required linear form for PB-constraints is explained in Chapter 4), as any of the $C_i.F_i$ could be an arithmetic expression over pseudo-boolean variables and map references rather than an integer value. This is also true of the explicit relational constraints that could be comparing arbitrary arithmetic expressions over pseudo-boolean values (variables or map references).

To reduce such constraints to the desired form, we first simplify each arithmetic expression as follows. First, expressions of the form `(-` E_1 E_2`)` are rewritten to `(+` E_1 `(-` E_2`))`. Next, expressions of the form `(-` E`)` are rewritten to the expression `(*` -1 E`)`. Finally, multiplication is distributed inside addition in the usual way (we obtain terms that are products of pseudo-boolean variables in the process). Once constants are collected and multiplied or added as appropriate, it results in an expression of the following form:

$$c + \sum_{i=1}^{n} a_i \times \prod_{j=1}^{n_i} v_{i,j}$$

where, c and each a_i is a constant integer value, each $v_{i,j}$ is a pseudo-boolean variable, and $\prod_{j=1}^{n_i} v_{i,j}$ expands into a product of n_i pseudo-boolean variables $(v_{i,1} * v_{i,2} * \ldots * v_{i,n_i})$.

Once we have simplified both of the expressions being compared in the relational constraint, we collect the integer constants on one side of the relation, and the terms involving variables on the other. This leaves us with an expression of the following form:

$$\sum_{i=1}^{n} a_i \times \prod_{j=1}^{n_i} v_{i,j} \; R \; c,$$

where R is a relation. But the formula still contains terms that are product of pseudo-boolean variables. However, note that $(v_{i,1} * v_{i,2} * \ldots * v_{i,n_i}) = 1$ in the integral context if and only if $v_{i,1} \wedge v_{i,2} \wedge \ldots \wedge v_{i,n_i}$ is true in the boolean context. Therefore, we can create a new pseudo-boolean variable x_i for each product term in the sum, and add the following formula to our list of boolean formulas:

$$x_i \Leftrightarrow (v_{i,1} \wedge v_{i,2} \wedge \ldots \wedge v_{i,n_i})$$

Finally, we substitute each x_i into the formula, giving us an expression of the desired form:

$$\sum_{i=1}^{n} a_i x_i \; R \; c,$$

In this fashion all field constraints and explicit relational constraints are converted by the back-end into a linear form suitable for the solver. Additionally, if the CoBaSA program has an objective constraint to be optimized, then it converts the corresponding arithmetic relational expression into a linear form in a similar manner.

Finally after processing all the map, field, relational, and objective constraints, the back-end converts the boolean constraints (or formulas) into CNF clauses. It uses the best known linear-time algorithm for doing the conversion [28]. Now that the back-end has generated an appropriate satisfiability and optimization problem in terms of linear PB-constraints, and CNF formulas, the control is passed on to the pseudo-boolean solver that prunes the search space in order to find an assignment of the pseudo-boolean variables that satisfies all constraints.

5.3 Pseudo-Boolean Solver: PBS

The CoBaSA system uses PBS [2, 1] (refer to Section 4.3 for more information on PBS) as solver for tackling the linear PB-constraints and CNF formulas. PBS accepts two files as input: a 'CNF' file and a 'PB' file. The back-end writes the CNF formulas in the CNF file, and it writes the PB-constraints in the PB file. After pruning the search space, PBS generates a raw output that just tells you for each pseudo-boolean variable in the satisfiability problem, if the variable is true or false. The output must be translated in terms of the user-defined mappings to obtain the CBSA solution. Additionally, if an optimization constraint is specified in the CoBaSA program, then PBS finds the optimum solution, which finally enables CoBaSA to present an optimized CBSA solution.

CHAPTER VI

INTEGRATED MODULAR AVIONICS

CoBaSA is designed for generic component assembly, meaning it can be utilized in several industries and domains. For instance it can be used within the avionics industry and the automotive industry, both of which have shifted focus from developing independent systems to developing modular systems. In this chapter, we focus on the avionics industry, by considering assembly of components within avionic systems. Note that current assembly techniques are not suitable for assembling avionics systems, as their focus on extra-functional properties is insufficient for the assembly of large reliable safety-critical distributed avionic systems (explained further in Chapters 1 and 2).

In the first section in this chapter, we introduce IMA concepts and describe the features of an IMA model. In next section we demonstrate how CoBaSA can be used to assemble IMA systems.

6.1 Introduction to IMA

Historically, avionic systems were built on a federated architecture consisting of black-boxes called Line Replaceable Units (LRUs), each of which was specifically designed to perform an individual function. For instance, an LRU could be an aircraft display unit, or one or more LRUs could work together to calculate and display the aircraft speed. In the federated platform each LRU has custom sensors, actuators, and dedicated displays; additionally communication between LRUs is extremely limited and hardly any resources are shared [66].

Due to growing complexity of equipment and advancements in technology, the avionics industry has now moved to implementing open architectures that employ highly-integrated digital avionics under software control. This approach, referred to as Integrated Modular

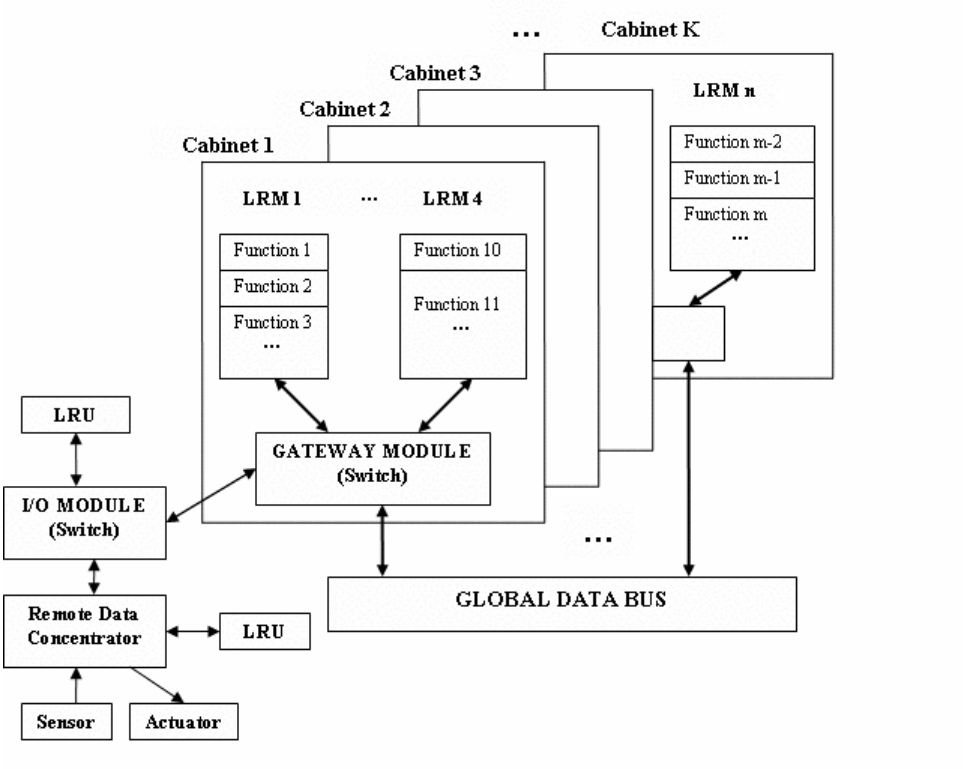

Figure 4: Diagrammatic Representation of Integrated Modular Avionics

Avionics (IMA) is based on modular design, generic resources and multiplexed communication buses; it has resulted in the development of smaller, lighter, cost-effective, and more reliable avionics equipment. In the integrated approach, the sensors, actuators, display devices and other avionic function equipments are connected to a central computing platform. For example, Boeing's next generation 787 aircraft takes advantage of a common core system that provides an open-standards computing platform on which more than 100 avionic functions can be realized in hardware and software [66].

The IMA platform is illustrated in Figure 4. It reflects the current standards in the avionics industry [48, 34, 51]. It is made up of cabinets containing sets of modules, called Line Replaceable Modules (LRMs). An LRM can be either a memory module or an Avionics Computing Resource (ACR) module. In the figure, we concern ourselves only with ACR modules. Each ACR is a computer which performs a variety of tasks (or avionic functions), each of which would have been performed by a separate LRU in older systems. Several functions that were originally implemented as independent LRUs, now face the possibility

of interacting with one another while sharing common resources (*e.g.*, , communication buses). To avoid unintended interactions, software applications running on the ACRs must follow strict guidelines for memory partitioning and timing.

The cabinets communicate with each other via a Gateway Module switch, which multiplexes communications through the Global Data Bus (over AFDX Ethernet or Fiber Channel) [51]. The gateway modules also handle communication with elements external to the cabinets. These include some LRUs, which take care of avionic functions that cannot be integrated into the cabinets.

Also included in the external elements are sensors and actuators which collect data and carry out commands throughout the aircraft. These elements are tied to the system using Remote Data Concentrators (RDCs) that regulate the data collected from and distributed to these elements. All the external elements communicate with the cabinets via additional I/O module switches. The data collected by a sensor is passed on as a message to an RDC, which through an I/O module switch passes the message to the gateway module switch, which through the global data bus passes the message to the specific avionic function (or software application) in an LRM in a cabinet. In a similar fashion, an avionic function in an LRM communicates its messages to an actuator to perform the required action.

6.2 CCAT in IMA Systems

Assembling components in IMA systems is tedious; there are typically about 4000 connections that have to be considered, in addition to an equal number of safety and reliability constraints imposed both at the component-level and system-level. Further complicating factors include scarcity of computation, memory, and bandwidth resources available onboard an aircraft. Since IMA is based on an open architecture scheme, it allows the system designer to shop around for COTS components. Furthermore, it is required that the IMA approach effortlessly support component changes and upgrades. This is where CCAT and CoBaSA comes into the picture; CCAT provides a mechanism for modeling the components and the constraints on their interaction in an intuitive way, and CoBaSA automatically finds an assembly that meets all the specified constraints. Since CoBaSA automates CBSA, it can

be repeatedly used as the system design evolves and components are changed or upgraded.

We now walk through a simple example of how components are assembled in an IMA system using CoBaSA. Consider first the task of mapping `m` avionic functions to `n` LRMs. (Assume that the components have been specified and the component instances have been created.) Typically `m` ranges from 200 to 400 and `n` ranges from 10 to 30. The required mapping or connection constraint appears as follows.

```
Map func-lrm func lrm
Constraint func-lrm ((processor-time-req, memory-req,
                      bandwidth-req-on-global-data-bus))
                    ((processor-time-available, memory-available,
                      bandwidth-available-on-global-data-bus))
```

The map `func-lrm` maps avionic function component instances to LRM component instances. The field constraints dictate that the avionic functions will draw their processing time requirements, memory requirements, and bandwidth requirements for the global data bus from the LRMs' available processing time, memory, and bandwidth respectively. The implicit constraint here is that each LRM must have enough of each resource to meet the needs of every consumer mapped to it. This constraint would ensure that the performance requirements of the avionic functions and LRMs are satisfied during component assembly. Another necessary mapping describes how the LRM component instances are to be arranged in the cabinets.

```
Map lrm-cabinet lrm cabinet
Constraint lrm-cabinet ((no-of-shelves-req)) ((no-of-shelves-provided))
For_all c in cabinet
   Sum l in lrm  lrm-cabinet(l,c)
   <= c.maximum-no-of-lrms-allowed
```

This code dictates that all the LRMs mapped to a given cabinet must be able to fit on that cabinet's shelf space. The `For_all` statement says that each cabinet cannot hold

more LRMs than it is allowed by its specification. Note that `lrm-cabinet(l,c)` is a map-reference that is true if LRM `l` is mapped to cabinet `c` (map references are explained in Section 5.1.3).

As we discussed in Section 5.1.4, CoBaSA allows for the simultaneous solving of several maps, even under the circumstances where the solution to one map depends on the solution to another. This becomes important in our constraints for RDCs, sensors, actuators, and I/O Modules.

```
For_all r in rdcs
  r.bandwidth-overhead-on-global-data-bus
  = Sum s in sen
    (* s.bandwidth-on-global-data-bus
       bandwidth-overhead-percent-for-sensor
       sen-rdc(s,r))
Map rdc-io-module rdcs io-switch
Constraint rdc-io-module
          ((bandwidth-overhead-on-global-data-bus,
            no-of-input-ports-to-switch-req,
            no-of-output-ports-to-switch-req))
          ((bandwidth-on-global-data-bus,
            no-of-input-ports, no-of-output-ports))
```

The above example assumes that the Global Data Bus bandwidth required from the IO-Module by the RDC depends on the bandwidth that the RDC provides for the sensors. Hence, the bandwidth for each RDC is computed to be the bandwidth required by the sensors mapped to it times an overhead percentage. As a result, the map between sensors and RDCs affects the map between the RDCs and IO-Modules. Since CoBaSA supports such chains of constraints, these interdependent maps can be solved appropriately.

For the purposes of safety and reliability considerations, partitions are built into the IMA model components by design [48]. These partitions ensure that the data of one partition does

not damage the data of another partition and that each partition holds its own execution window on the LRM to which it is mapped. In addition to these partitions, there are stringent safety and separation requirements that are imposed on the placements of the avionic functions within the LRMs and the placement of the LRMs within the cabinets. These requirements can be modeled as separation constraints in CoBaSA and these can be used by the solver to obtain an allocation that meets the critical safety requirements imposed on the aircraft. For example, consider the following.

```
for_all i in lrm {(func-lrm(func[1], i)
                   implies ((not func-lrm(func[2], i)) and
                            ((not func-lrm(func[3], i)) and
                             (not func-lrm(func[4], i)))))}
for_all i in lrm {(func-lrm(func[2], i)
                   implies ((not func-lrm(func[3], i)) and
                            (not func-lrm(func[4], i))))}
for_all i in lrm {(func-lrm(func[3], i)
                   implies (not func-lrm(func[4], i)))}
```

The above `for_all` statements specify that the first 4 avionic functions (these are redundant copies of a function) must all map to different LRMs. Constraints along the same lines can be used to express the intricate separation constraints that exist among the IMA components.

Reduction of wiring costs is an important consideration when designing cost-effective IMA systems. Intuitively wiring costs can be reduced if a component is mapped to the nearest appropriate component, rather than a component that is further away from it in terms of wire length. Such an optimization constraint could be expressed as follows.

```
Minimize (+ Sum r in rdcs
            (* rdc-lru-port-1(lru-port-1,r)
               (let ((lx lru-port-1.x-pos) (ly lru-port-1.y-pos)
                     (lz lru-port-1.z-pos) (rx r.x-pos)
                     (ry r.y-pos) (rz r.z-pos))
```

```
        _(ceiling (sqrt (+ (expt (- rx lx) 2)
                           (expt (- ry ly) 2)
                           (expt (- rz lz) 2))))_))
    ...) <= max-allowed-sum-of-distances
```

The optimization constraint minimizes the distance between an LRU port and the RDC port to which it is connected. The constraint specifies the distance expression for `lru-port-1`, similar expressions can be written for all the concerned ports. By minimizing the sum of the distance expressions, we can minimize the wiring costs of the IMA system.

CHAPTER VII

EVALUATION

Since our work on CoBaSA is in collaboration with Boeing, we implemented CoBaSA on real IMA models, which will be used in the forthcoming Boeing 787 aircraft. Unfortunately, we could find no publicly available models for systems in other domains to evaluate CoBaSA with. Nevertheless, since CoBaSA is developed for generic CBSA and it is not fine-tuned for any domain specific features; it can be applied in several industrial domains other than just the avionics domain. In this chapter, first we will describe the Boeing IMA models and then go one to describe how CoBaSA simplifies component assembly for Boeing.

The Boeing IMA models are very complex and are a part of an evolving design. They involve hundreds of components and equally many constraints. For such problems, Boeing requires half a man-month to create a CoBaSA model from well-understood data, and over a man-week to verify that a given configuration satisfies the constraints. Boeing has requested that we do not disclose how long these problems take to solve using conventional methods, but it suffices to say that finding the optimal assembly is significantly more difficult than creating the problem or verifying a given solution. By expressing the IMA models as CoBaSA programs, we can solve Boeing IMA problems in minutes, furthermore we have developed a checker that verifies CoBaSA solution against the assembly constraints. Over the course of our work with Boeing, CoBaSA was able to easily handle what Boeing described as "serious architecture changes."

7.0.1 IMA Models

The IMA models that we obtained from Boeing, focused on the assembly of black-box components like avionic functions, LRMs, cabinets, *etc.*, as shown in Table 1. The nature of the components is same across all the versions given in the table, but the number of components changes from one version to another. The change in the number of components indicates

Table 1: Description of IMA Model Versions

	Version 1	Version 2	Version 3
# Cabinets	2	2	2
# LRMs	16	16	22
# Avionic Functions	237	257	245
# Linked Memories	70	88	88
# Constraints	224	268	271

that the IMA versions are architecturally different. We have already seen the simplified descriptions of components like cabinets, LRMs, and avionic functions, in Chapter 6, but in real Boeing IMA models, these components are much more complex; they have fields that incorporate worst-case execution time, context switching time, I/O time, latency, network jitter, context switching time, cache flushing time, memory latencies, and so on. The new components that we have not seen before are the linked memories. Linked memories are memories, allocated within the LRMs, that are linked to either a specific avionic function or a set of avionic functions. The avionic functions and linked memories, both obtain their memory requirements from the LRM on which they are allocated, but in addition each avionic function has access to the linked memory to which it is linked. Hence, it is desirable to allocate a linked memory to the same LRM, to which the avionic functions that link to that linked memory are allocated.

Additionally, Table 1 gives the number of constraints that are present in the 3 versions. The nature of the constraints remains almost same, but the total number of constraints changes from one version to another. In Table 1 '# Constraints' expresses the total constraint count, and it includes map constraints, field constraints, reliability constraints, and safety constraints. The map and field constraints are very similar to ones described in Chapter 6, but the reliability and safety constraints are far more complicated. Furthermore, the constraints between the linked memories and the avionic functions that they are linked to are quite intricate. For instance, when allocating a memory region to an LRM, it is necessary to consider the separation constraints among the avionic functions within the set of avionic functions that the linked memory is linked to.

Table 2: Summary of Experimental Results

Version		CNF			PB		Results			
#	File Size	File Size	Vars	Clauses	File Size	Constraints	Parsing	Compiling	Solving	Result
1	45.9K	24.7K	2704	1768	133.2K	371	12.83s	3.72s	0.05s	SAT
2	59.7K	49.9K	2952	3748	138.3K	409	29.21s	6.55s	0.00s	UNSAT
3	57.6K	66.0K	3971	4861	180.9K	421	29.12s	13.47s	0.15s	SAT

7.0.2 Experimental Results

The experimental results that we obtained is summarized in Table 2. The table includes timing results and size descriptions for the various files. From the table we know that for version 1, the IMA model is written as a 45.9KB CoBaSA program; this program is parsed and type checked within 12.83 seconds, and then within 3.72 seconds it is translated into two files (CNF file and PB file) of sizes 24.7KB and 133.2KB. The generated files are given as input to the pseudo-boolean solver that finds a satisfiable solution in 0.05 seconds. From the CNF and PB statistics, we can infer the complexity of the input files that are given to the pseudo-boolean solver. Note that the CNF portions of the pseudo-boolean problems are relatively small, but the linear constraints portions are significantly larger. Parsing and compiling take up the bulk of running time, due to the fact that our compilation code has not been optimized for performance.

Out of the 3 versions, it is version 2 that results in an unsatisfiable solution. For this version, the solver realizes through initial conflict analysis that the problem is unsatisfiable. This means that for version 2, the IMA model cannot be assembled such that all the constraints are satisfied. The solutions given by PBS for versions 1 and 3, after conversion to human readable form, describe how to successfully assemble the components within the IMA models. These solutions were verified by a checker to ensure that the component assembly satisfies all the constraints; moreover Boeing engineers confirmed that the component assembly satisfied their requirements. Note that none of the problems, satisfiable or unsatisfiable, took longer than 45 seconds to complete. This is in sharp contrast to the much greater time taken by Boeing to find satisfying assignments using conventional assembly techniques.

With such low solving times, we were able to make significant architectural changes and

find new configurations for the models. For example, the original CoBaSA problem that we received for version 3 of the model contained only 16 LRMs. There was no satisfying assignment, so we increased the number of LRMs to 18 and so on until we found the satisfying assignment using 22 LRMs. This entire exercise took only several minutes using CoBaSA, but would have taken weeks if done using Boeing's current methods.

CHAPTER VIII

FUTURE WORK AND CONCLUSION

Component-based software development and design has the potential to significantly impact software development in the large, but several technical challenges remain. One of them is automated CBSA of systems that have complex assembly constraints. We presented CCAT as an assembly technique for systems that have a large number of complex assembly constraints. Based on CCAT, we developed CoBaSA as a powerful framework that includes an expressive language for declaratively specifying component functional and extra-functional properties, component interfaces, system-level and component-level connection, performance, reliability, safety, and optimization constraints. We developed algorithms to convert problems expressed in our language to pseudo-boolean problems, allowing us to leverage the recent advances in SAT solving technology. Based on our empirical evaluation of CoBaSA in the avionics domain, we can conclude that CoBaSA can scale to problems of industrial interest and be applied to a wide class of CBSA problems.

For future work, we plan to apply CoBaSA in the automotive domain, and the software package configuration domain. The automotive and avionic industry share similar concerns, and hence it is likely that CoBaSA will be very useful in the automotive industry. Software package configuration system, which greatly simplify system administration, treat packages as components that have constraints among them. CoBaSA has the scope of configuring packets efficiently, as it provides an intuitive way of expressing complex package dependencies and conflicts. In addition to extending the applications of CoBaSA, the framework itself can be improved with some algorithmic extensions. These include decomposing the assembly problem into several independent sub-problems, and using a more efficient encoding scheme that would reduce the number of pseudo-boolean variables in the satisfiability problem, and lastly using a more efficient pseudo-boolean solver that would speed up CoBaSA when it faces difficult CBSA instances.

APPENDIX A

COBASA LANGUAGE SYNTAX AND SEMANTICS

A.1 Syntax

A.1.1 Symbols

Explanation for the symbols used in describing the syntax. The Grammar is case insensitive.

Bold Characters	:	Keywords or symbols that appear in the language.
$^{+}$	:	Represents one or more.
*	:	Represents zero or more.
^{n}	:	Represents exactly n.
$^{?}$	:	Represents optional characters/symbols.
()	:	Represents grouping within the constructs.

A.1.2 Preliminaries

<Prog> ::= <Decls> <Constr> (<Objective>)$^{?}$

<Num> ::= (0-9)

<Nat> ::= <Num>$^{+}$

<Int> ::= -$^{?}$<Nat>

<Range> ::= [<Int> .. <Int>]

<Alph> ::= (a-z | A-Z)

<Name> ::= <Alph> (<Alph> | <Num> | - | _)*

<Str> ::= `` <Name> ''

<Rel> ::= < | > | <= | >= | =

<Bool_op> ::= not | and | or | implies | iff

<Single_Line_Comment> ::= //

<Block_Comment> ::= #| |#

A.1.3 Declarations

<Decls> ::= (<Enum_Decl> | <Entity_Decl> | <Var_Decl> | <Const_Decl> | <Assign>)*

<Const_Ref> ::= <Name> <Array_Dim>? | <Name> <Array_Dim>?.<Const_Ref> |
(<Name>)<Const_Ref> | <Const_Ref>.dim[<Nat>] | <Const_Ref>.rank

<Array_Dim> ::= ([<Nat> | <Const_Ref>])+

<Base_Const> ::= Null | True | False | <Str> | <Int> | <Lterm> | <Const_Ref>

<Ent_Const> ::= <Name> <UEnt_Const>

<UEnt_Const> ::= {(;<UConst>?)*}

<Array_Const> ::= <Type> <Array_Dim> = <UArray_Const>

<UArray> ::= [(<Base_Const> | <UEnt_Const>) (, (<Base_Const> | <UEnt_Const>))*]

<UArray_Const> ::= <UArray> | [<UArray_const>+]

<Const> ::= <Base_Const> | <Ent_Const> | <Array_Const>

<UConst> ::= <Base_Const> | <UEnt_Const> | <UArray_Const>

<Type> ::= (String | Int | Bool | <Range> | <Name>) (<Array_Dim>)?

<Field> ::= (;<Name> (<Const> | <Type>))*

<Variable> ::= (;<Type> <Name> (=<UConst>)?)+

<Cnst> :: (;<Name> <Const>)+

<Enum_Decl> ::= Enum <Name> (<Name> (, <Name>)*)

<Entity_Decl> ::= Entity <Name> (Extends <Name>)? {<Field>}

<Var_Decl> ::= Var <Variable>

<Const_Decl> ::= Const <Cnst>

<Assign> ::= Assign <Const_ref> = <UConst>

A.1.4 Constraints

<Constr> ::= (<Map_Def> | <Constraint> | <For_all_exp> | <Assign>)*

<LCRef> ::= <Const_Ref> | (<Const_Ref> (, <Const_Ref>)$^+$)

<LCField_ref> ::= (((<Const_Ref> | (<Name>)_.<Const_Ref>)
(, (<Const_Ref> | (<Name>)_.<Const_Ref>))*))$^+$)

<Map_Ref> ::= <Name> (<Const_Ref> (,<Const_Ref>)$^?$)

<Lterm> ::= (let (((<Name> (<Const_Ref> | <Map_Ref>)))$^+$) _<Lisp_Term>_)

<Lisp_Term> ::= Lisp expression with free variables that are assigned values by let within Lterm.

<Term> ::= Sum <Name> in (<Const_Ref> | <Nat>) <Term> | (+ <Term>$^+$) | (* <Term>$^+$) |
(- <Term> <Term>$^?$) | <LTerm> | <Const_Ref> | <Int> | <Map_Ref>

<Bool_exp> ::= <Const_Ref> | <Map_Ref> | <LTerm> | (<Bool_exp>) | Not <Bool_exp> |
<Bool_exp> <Bool_op> <Bool_exp>

<Map_Def> ::= Map <Name> <LCRef> <LCRef>

<Constraint> ::= Constraint <Name><LCField_ref> <LCField_ref>

<For_all_exp> ::= For_all <Name> in (<Const_Ref> | <Nat>) { <For_all_exp> } |
<For_all_exp> and <For_all_exp> | <Term> <Rel> <Term> | <Bool_Exp>

A.1.5 Objective

<Objective> ::= (Maximize | Minimize) <Term> <Rel> <Term>

A.2 Semantics

A.2.1 Preliminaries

ϵ denotes the variable environment, and ζ denotes the empty string. The set $PBProb$ is a boolean formula whose free variables all range over the set $\{0, 1\}$, or *Error*.

$Names = \{n \mid n$ is derived using Name$\}$.

$Names' = Names \cup \{\zeta\}$.

$Strings = \{s \mid s$ is derived using Str$\}$.

$Operators = \{<, >, <=, >=, =, +, *, -, \texttt{or}, \texttt{and}, \texttt{implies}, \texttt{iff}\}$.

$Bool\_Value = \{\texttt{True}, \texttt{False}\}$.

$Int\_Bool\_Value = \mathbb{Z} \cup Bool\_Value$.

$Field\_func : Names' \times Names \rightarrow (Type' \times Value')$.

$Entities = \{\langle N_1, N_2, F\rangle \mid N_1 \in Names,\ N_2 \in Names',\ F : Field\_func\}$.

$Enums = \{\langle N, E\rangle \mid N \in Names,\ E : Names \times Names \rightarrow \mathbb{N}\}$.

$Objects = \{\langle T, O\rangle \mid T \in ObjectType,\ O : Names' \times Names \rightarrow Value'\}$.

$BaseVal = \{\langle T, V\rangle \mid T \in BaseType,\ V : Value'\}$.

$Pointer = \{\langle ref.v\rangle \mid v \in Value\}$ (Pointer to v and any changes made to $ref.v$ would change v).

$Arrays = \{[v_0\ v_1\ \ldots\ v_n] \mid n \in \mathbb{N},\ \forall 0 \leq i \leq n,\ v_i \in Value\ \wedge\ \forall i\ \mathcal{VT}\ (v_i)$ is same$\}$.

$Lists = \{(v_0\ v_1\ \ldots\ v_n) \mid n \in \mathbb{N},\ \forall 0 \leq i \leq n,\ v_i \in Value\}$.

$Maps = \{\langle C, R, M\rangle \mid\ C, R \in Lists,\ M : Value \rightarrow Value\}$.

$Value = \{\texttt{Null}\} \cup Int\_Bool\_Value \cup Strings \cup Arrays \cup BaseVal \cup Objects$.

$Value' = Value \cup \{\perp\} \cup Pointer$.

$Value^{\#} = Value' \cup \{Error\}$.

$Values = Value \cup Maps$.

$Values' = Value' \cup Maps$.

$Values^{\#} = Value^{\#} \cup Maps$.

$BaseType = \{\langle T\rangle \mid T \in \{\texttt{String}, \texttt{Int}, \texttt{Bool}\}\} \cup \{\langle i_1, i_2\rangle \mid i_1, i_2 \in \mathbb{Z},\ i_1 < i2\}$.

$ObjectType = \{\langle N_1, N_2, F\rangle \mid N_1 \in Names,\ N_2 \in Names',\ F : Field\_func\}$.

$BaseArrayType = \{\langle T, (d_1, d_2, \ldots, d_n), S\rangle \mid T \in BaseType,\ n \in \mathbb{N},\ \forall 1 \leq i \leq n,\ d_i \in \mathbb{N},\ S = (d_1 * d_2 * \cdots * d_n) - 1\}$.

$ObjectArrayType = \{\langle T, (d_1, d_2, \ldots, d_n), S\rangle \mid T \in ObjectType,\ n \in \mathbb{N},\ \forall 1 \leq i \leq n,\ d_i \in \mathbb{N},\ S = (d_1 * d_2 * \cdots * d_n) - 1\}$.

$ObjectsType = ObjectType \cup ObjectArrayType$.

$ArrayType = BaseArrayType \cup ObjectArrayType$.

$Type = BaseType \cup ObjectType \cup ArrayType$.

$Type' = \{\perp, Error\} \cup Type$.

$MapType = \{\langle C, R, M\rangle \mid\ C, R \in Lists,\ M : Value \rightarrow Value\}$.

$nth : Arrays \times \mathbb{N} \rightarrow Value$ (returns the n^{th} value of an array).

$sts : Arrays \rightarrow 2^{Value}$ (turns an array into a set, and a singleton into a singleton set).

fld_cnstr : $Maps \times$ LCField_ref $\times$ LCField_ref $\rightarrow PBProb$ (generates the field constraints for the map).

$map\_cnstr : Names \times Value \times Value \rightarrow PBProb$ (generates the map constraints for the map).

$append\_fields : Field\_func \times Field\_func \rightarrow Field\_func$ (appends the extended entity *field_func* to the entity *field_func*).

$\mathcal{D}$: Decls $\times$ $(Names \rightarrow Value^{\#}) \times$ Decls$^{*} \rightarrow PBProb$.

$\mathcal{V}$: Variable $\times$ $(Names \rightarrow Value^{\#}) \times$ Variable$^{*} \rightarrow PBProb$.

$\mathcal{G}$: Cnst $\times$ $(Names \rightarrow Value^{\#}) \times$ Cnst$^{*} \rightarrow PBProb$.

$\mathcal{FL}$: Field $\times$ $Names \times Names' \times Field\_func \times (Names \rightarrow Value^{\#}) \times$ Field$^{*} \rightarrow PBProb$.

$\mathcal{EM}$: $Names \rightarrow (Names \times Names \rightarrow \mathbb{N})$.

$\mathcal{C}$: Constr $\times$ $(Names \rightarrow Values^{\#}) \times$ Constr$^{*} \rightarrow PBProb$.

$\mathcal{T}$: Term $\times$ $(Names \rightarrow Values^{\#}) \rightarrow Values^{\#}$.

$\mathcal{B}$: Bool_exp $\times$ $(Names \rightarrow Values^{\#}) \rightarrow PBProb$.

$\mathcal{CR}$: Const_Ref $\times$ $(Names \rightarrow Value^{\#}) \rightarrow Value^{\#}$.

$\mathcal{MR}$: Map_Ref $\times$ $(Names \rightarrow Values^{\#}) \rightarrow Bool\_Value \cup \{\perp,\ Error\}$.

$\mathcal{LT}$: Lterm $\times$ $(Names \rightarrow Value^{\#}) \rightarrow Type' \times Value^{\#}$.

$\mathcal{TP}$: Type $\times$ $(Names \rightarrow Value^{\#}) \rightarrow Type'$.

$\mathcal{CT}$: Const $\times$ $(Names \rightarrow Value^{\#}) \rightarrow (Type' \times Value^{\#})$.

$\mathcal{BCT}$: Base_Const $\times$ $(Names \rightarrow Value^{\#}) \rightarrow (Type' \times Value^{\#})$.

$\mathcal{ECT}$: Ent_Const $\times$ $(Names \rightarrow Value^{\#}) \rightarrow (Type' \times Value^{\#})$.

$\mathcal{ACT}$: Array_Const $\times$ $(Names \rightarrow Value^{\#}) \rightarrow (Type' \times Value^{\#})$.

$\mathcal{UCT}$: UConst $\times$ $Type \times (Names \rightarrow Value^{\#}) \rightarrow Value^{\#}$.

$\mathcal{UBCT}$: Base_Const $\times$ $Type \times (Names \rightarrow Value^{\#}) \rightarrow Value^{\#}$.

$\mathcal{UECT}$: UEnt_Const $\times$ $Type \times (Names \rightarrow Value^{\#}) \rightarrow Value^{\#}$.

$\mathcal{UA}$: UArray $\times$ $Type \times (Names \rightarrow Value^{\#}) \rightarrow Value^{\#}$.

$\mathcal{UACT}$: UArray_Const $\times$ $Type \times (Names \rightarrow Value^{\#}) \rightarrow Value^{\#}$.

$\mathcal{O}$: Objective $\times$ $(Names \rightarrow Values^{\#}) \rightarrow PBProb$.

$\mathcal{R}$: Range $\rightarrow Type'$.

$\mathcal{OP}$: $Operator \rightarrow Operator$.

$\mathcal{VT}$: $Values \rightarrow Type'$.

$\mathcal{FC}$: $Names \rightarrow (Objects \times Names \times Objects \rightarrow Names)$ (given a mapping name, consumer, a field of that consumer, and a provider, returns the name of the field of that provider from which the consumer field would draw if the consumer were mapped to the provider).

$\mathcal{IND}$: Array_Dim $\times$ $(Names \rightarrow Value^{\#}) \rightarrow Lists \times Value^{\#}$. ($Lists$ is the list of indices and $Value^{\#}$ is the row major index corresponding to the indices.)

$\mathcal{DIM}$: Array_Dim $\times$ $(Names \rightarrow Value^{\#}) \rightarrow Lists \times Value^{\#}$. ($Lists$ is the list of dimensions and $Value^{\#}$ is the row major index of the last element.)

A.2.2 Declarations

$$\mathcal{D}\,[\![\texttt{Enum}\ N\ E]\!]\ \epsilon\ d \quad = \quad \begin{cases} \mathit{Error} & \text{if } \epsilon.N \\ \mathcal{D}\,[\![\mathit{first}(d)]\!]\ \epsilon'\ \mathit{rest}(d) & \text{otherwise.} \end{cases}$$
$$\text{where } e = \mathcal{EM}\,[\![E]\!]\ \epsilon,$$
$$\epsilon' = \epsilon[N \mapsto \langle N, e\rangle].$$

$$\mathcal{D}\,[\![\texttt{Entity}\ N_1\ \texttt{Extends}\ N_2\ \{\ F\ \}]\!]\ \epsilon\ d \quad = \quad \begin{cases} \mathit{Error} & \text{if } (\epsilon.N_1\ \vee\ \epsilon.N_2 = \ \bot) \\ \mathcal{D}\,[\![\mathit{first}(d)]\!]\ \epsilon'\ \mathit{rest}(d) & \text{otherwise.} \end{cases}$$
$$\text{where } f = \bot,\ \mathcal{FL}\,[\![first(F)]\!]\ N_1\ N_2\ f\ \epsilon'\ rest(F),$$
$$\langle e_1, e_2, f_2\rangle = \epsilon.N_2,\ f = append\_fields(f, f_2, N_2),$$
$$\epsilon' = \epsilon[N_1 \mapsto \langle N_1, N_2, f\rangle].$$

$$\mathcal{D}\,[\![\texttt{Entity}\ N\ \{\ F\ \}]\!]\ \epsilon\ d \quad = \quad \begin{cases} \mathit{Error} & \text{if } \epsilon.N \\ \mathcal{D}\,[\![\mathit{first}(d)]\!]\ \epsilon'\ \mathit{rest}(d) & \text{otherwise.} \end{cases}$$
$$\text{where } f = \bot,\ \mathcal{FL}\,[\![first(F)]\!]\ N_1\ f\ \zeta\ \epsilon'\ rest(F),$$
$$\epsilon' = \epsilon[N \mapsto \langle N, \zeta, f\rangle].$$

$$\mathcal{D}\,[\![\texttt{Var}\ V]\!]\ \epsilon\ d \quad = \quad \mathcal{V}\,[\![first(V)]\!]\ \epsilon\ rest(V); d$$

$$\mathcal{D}\,[\![\texttt{Const}\quad \texttt{C}]\!]\ \epsilon\ d \quad = \quad \mathcal{G}\,[\![first(C)]\!]\ \epsilon\ rest(C); d$$

$$\mathcal{D}\,[\![\texttt{Assign}\ CR\ \texttt{=}\ O]\!]\ \epsilon\ d \quad = \quad \begin{cases} \mathit{Error} & \text{if } ((cr \neq \bot) \vee (t_1 \neq t_2)) \\ \mathcal{D}\,[\![\mathit{first}(d)]\!]\ \epsilon'\ \mathit{rest}(d) & \text{otherwise.} \end{cases}$$
$$\text{where } cr = \mathcal{CR}\,[\![CR]\!]\ \epsilon,\ t_1 = \mathcal{VT}\,(cr)\,,\ o = \mathcal{UCT}\,[\![O]\!]\ t\ \epsilon,\ t_2 = \mathcal{VT}\,(o)\,,$$
$$\text{if } (t_2 \in \mathit{ObjectsType}\ \wedge\ cr \text{ is a } \mathit{Pointer}),$$
$$\text{then } o \text{ is a } \mathit{Pointer} \text{ pointing to object pointed by } cr,$$
$$\text{else } \epsilon' = set(CR, o, \epsilon)$$

$$\mathcal{FL}\,[\![;N\ TC]\!]\ n_1\ n_2\ fl\ \epsilon\ f \quad = \quad \begin{cases} \mathit{Error} & \text{if } N \in dom.fl \\ \mathcal{FL}\,[\![first(f)]\!]\ n_1\ n_2\ fl'\ \epsilon\ rest(f) & \text{otherwise.} \end{cases}$$
$$\text{where if } TC \text{ is syntactically Type, } t = \mathcal{TP}\,[\![TC]\!]\ \epsilon,\ v = \bot,$$
$$\text{if } TC \text{ is syntactically Const, } \langle t, v\rangle = \mathcal{CT}\,[\![TC]\!]\ \epsilon,$$
$$fl' = [fl(\zeta, N) \mapsto \langle t, v\rangle].$$

$$\mathcal{V}\,[\![;T\ N]\!]\ \epsilon\ v \quad = \quad \begin{cases} \mathit{Error} & \text{if } (\epsilon.N \vee t \notin \mathit{Type}) \\ \mathcal{V}\,[\![\mathit{first}(v)]\!]\ \epsilon'\ \mathit{rest}(v) & \text{otherwise.} \end{cases}$$

where $t = \mathcal{TP}\,[\![T]\!]\ \epsilon'$,

if $t \in \mathit{ObjectType},\ o = \langle t, \perp\rangle$,

if $t \in \mathit{ObjectArrayType},\ t = \langle ty, d, s\rangle,\ o = [\langle t, \perp_0\rangle, \ldots, \langle t, \perp_{s-1}\rangle]$,

otherwise $o = \perp$,

$\epsilon' = \epsilon[N \mapsto \langle t, o\rangle]$.

$$\mathcal{V}\,[\![;T\ N\ \texttt{=}\ C]\!]\ \epsilon\ v \quad = \quad \begin{cases} \mathit{Error} & \text{if } (\epsilon.N \vee t \notin \mathit{Type}) \\ \mathcal{V}\,[\![\mathit{first}(v)]\!]\ \epsilon'\ \mathit{rest}(v) & \text{otherwise.} \end{cases}$$

where $t = \mathcal{TP}\,[\![T]\!]\ \epsilon'$,

$o = \mathcal{UCT}\,[\![C]\!]\ t\ \epsilon$,

$\epsilon' = \epsilon[N \mapsto \langle t, o\rangle]$.

$$\mathcal{G}\,[\![;N\ \texttt{=}\ C]\!]\ \epsilon\ c \quad = \quad \begin{cases} \mathit{Error} & \text{if } \epsilon.N \\ \mathcal{G}\,[\![\mathit{first}(c)]\!]\ \epsilon'\ \mathit{rest}(c) & \text{otherwise.} \end{cases}$$

where $\langle t, o\rangle = \mathcal{CT}\,[\![C]\!]\ \epsilon$,

$\epsilon' = \epsilon[N \mapsto \langle t, o\rangle]$.

$$\mathcal{TP}\,[\![T]\!]\ \epsilon \quad = \quad \begin{cases} \mathit{Int} & \text{if } T = \texttt{Int}, \\ \mathit{Bool} & \text{if } T = \texttt{Bool}, \\ \mathit{String} & \text{if } T = \texttt{String}, \\ \mathcal{R}\,[\![T]\!] & \text{if } T = \text{is syntactically Range}, \\ \epsilon.T & \text{if } \epsilon.T \in \mathit{ObjectType}, \\ \mathit{Error} & \text{otherwise.} \end{cases}$$

$$\mathcal{TP}\,[\![T\ D]\!]\ \epsilon \quad = \quad \begin{cases} \mathit{Error} & \text{if } t \in \mathit{Error}, \perp, \\ \langle t, d, s\rangle & \text{otherwise.} \end{cases}$$

where $t = \mathcal{TP}\,[\![T]\!]\ \epsilon,\ \langle d, s\rangle = \mathcal{DIM}\,[\![D]\!]\ \epsilon$.

$$\mathcal{CT}\,[\![C]\!]\ \epsilon \quad = \quad \begin{cases} \mathcal{BCT}\,[\![C]\!]\ \epsilon & \text{if } C \text{ is syntactically Base\_Const}, \\ \mathcal{ECT}\,[\![C]\!]\ \epsilon & \text{if } C \text{ is syntactically Ent\_Const}, \\ \mathcal{ACT}\,[\![C]\!]\ \epsilon & \text{if } C \text{ is syntactically Array\_Const}, \\ \mathit{Error} & \text{otherwise.} \end{cases}$$

$$\mathcal{BCT}\,[\![C]\!]\ \epsilon \quad = \quad \begin{cases} \langle Null, Null\rangle & \text{if } C = Null, \\ \langle Bool, True\rangle & \text{if } C = True, \\ \langle Bool, False\rangle & \text{if } C = False, \\ \langle Int, C\rangle & \text{if } C \in \mathbb{Z}, \\ \langle String, C\rangle & \text{if } C \text{ is syntactically Str}, \\ \mathcal{LT}\,[\![C]\!]\ \epsilon & \text{if } C \text{ is syntactically Lterm}, \\ \langle t, v\rangle, \text{ where } v = \mathcal{CR}\,[\![C]\!]\ \epsilon,\ t = \mathcal{VT}\,(v) & \text{if } C \text{ is syntactically Const\_Ref} \\ Error & \text{otherwise.} \end{cases}$$

$$\mathcal{ECT}\,[\![N\ E]\!]\ \epsilon \quad = \quad \begin{cases} Error & \text{if } ((\epsilon.N \in \{\bot, Error\}) \vee (\epsilon.N \notin ObjectType)) \\ \langle t, o\rangle & \text{otherwise.} \end{cases}$$
$$\text{where } t = \epsilon.N,\ o = \mathcal{UECT}\,[\![E]\!]\ t\ \epsilon$$

$$\mathcal{ACT}\,[\![T\ D\ = A]\!]\ \epsilon \quad = \quad \begin{cases} Error & \text{if } t \notin Type \\ \langle t', o\rangle & \text{otherwise.} \end{cases}$$
$$\text{where } t = \mathcal{TP}\,[\![T]\!]\ \epsilon,\ \langle d, s\rangle = \mathcal{DIM}\,[\![D]\!]\ \epsilon,$$
$$t' = \langle t, d, s\rangle, o = \mathcal{UACT}\,[\![A]\!]\ t\ \epsilon$$

$$\mathcal{UCT}\,[\![C]\!]\ type\ \epsilon \quad = \quad \begin{cases} \mathcal{UBCT}\,[\![C]\!]\ type\ \epsilon & \text{if } C \text{ is syntactically Base\_Const}, \\ \mathcal{UECT}\,[\![C]\!]\ type\ \epsilon & \text{if } C \text{ is syntactically UEnt\_Const}, \\ \mathcal{UACT}\,[\![C]\!]\ type\ \epsilon & \text{if } C \text{ is syntactically UArray\_Const}, \\ Error & \text{otherwise.} \end{cases}$$

$$\mathcal{UBCT}\,[\![C]\!]\ type\ \epsilon \quad = \quad \begin{cases} Null & \text{if } C = Null\ \wedge\ type = Null, \\ True & \text{if } C = True\ \wedge\ type = Bool, \\ False & \text{if } C = False\ \wedge\ type = Bool, \\ C & \text{if } C \in \mathbb{Z}\ \wedge\ type = Int, \\ C & \text{if } C \text{ is syntactically Str}\ \wedge\ type = String, \\ l, \text{ where } \langle t, l\rangle = \mathcal{LT}\,[\![C]\!]\ \epsilon & \text{if } C \text{ is syntactically Lterm}\ \wedge\ type = t, \\ v, \text{ where } v = \mathcal{CR}\,[\![C]\!]\ \epsilon,\ t = \mathcal{VT}\,(v) & \text{if } C \text{ is syntactically Const\_Ref}\ \wedge\ type = t, \\ Error & \text{otherwise.} \end{cases}$$

$$\mathcal{UECT}\,[\![\{\,;E_1\ \ldots\,;E_l\}]\!]\ type\ \epsilon \quad = \quad \begin{cases} Error & \text{if } (l \neq s \vee (\exists i :: (c_i \neq \bot)\ \wedge\ (c_i \neq v_i))) \\ o & \text{otherwise.} \end{cases}$$

where $type = \langle m_1, m_2, f\rangle,\ s = card.dom.f,$

$\forall 1 \leq i \leq s\ o(n_{i1}, n_{i2}) \mapsto v_i$

where $\langle t_i, c_i\rangle = f(n_{i1}, n_{i2}),$

$v_i = \mathcal{UCT}\,[\![E_i]\!]\ \langle t_i, \bot\rangle\ \epsilon,$ if $(E_i \neq \zeta)\ \wedge\ (c_i = \bot),$

$v_i = \mathcal{UCT}\,[\![E_i]\!]\ \langle t_i, c_i\rangle\ \epsilon,$ if $(E_i \neq \zeta)\ \wedge\ (c_i \neq \bot)\ \wedge\ (c_i = v_i),$

$v_i = c_i,$ if $(E_i = \zeta)\ \wedge\ (c_i \neq \bot),$

$v_i = \bot,$ if $(E_i = \zeta)\ \wedge\ (c_i = \bot).$

$$\mathcal{UA}\,[\![\texttt{[}A_0,\ \ldots, A_l\texttt{]}]\!]\ type\ \epsilon \quad = \quad \begin{cases} Error & \text{if } ((l \neq s) \vee (len(d) \neq 1)), \\ [\mathrm{o}_0\ \ldots\ \mathrm{o}_l] & \text{otherwise.} \end{cases}$$

where $\langle t, d, s\rangle = type,$

$\forall 0 \leq i \leq l\ o_i = \mathcal{UBCT}\,[\![A_i]\!]\ t\ \epsilon,$ if $t \in BaseType,$

$\forall 0 \leq i \leq l\ o_i = \mathcal{UECT}\,[\![A_i]\!]\ t\ \epsilon,$ if $t \in ObjectType.$

$$\mathcal{UACT}\,[\![A]\!]\ type\ \epsilon \quad = \quad \begin{cases} Error & \text{if } len(d) \neq 1, \\ \mathcal{UA}\,[\![A]\!]\ type\ \epsilon & \text{otherwise.} \end{cases}$$

$$\mathcal{UACT}\,[\![\texttt{[}A_0\ \ldots A_l\texttt{]}]\!]\ type\ \epsilon \quad = \quad \begin{cases} Error & \text{if } (l \geq first(d) \vee len(d) \leq 1) \\ [\mathrm{o}_0\ \ldots\ \mathrm{o}_l] & \text{otherwise.} \end{cases}$$

where $\langle t, d, s\rangle = type,$

$\forall 0 \leq i \leq l\ o_i = \mathcal{UA}\,[\![A_i]\!]\ \langle t, rest(d), s\rangle\ \epsilon,$ if $len(d) = 2,$

$\forall 0 \leq i \leq l\ o_i = \mathcal{UACT}\,[\![A_i]\!]\ \langle t, rest(d), s\rangle\ \epsilon,$ if $len(d) \geq 3.$

$$\mathcal{CR}\,[\![N]\!]\ \epsilon \quad = \quad \begin{cases} Error & \text{if } \epsilon.N \in \{Error, \bot\} \\ v & \text{otherwise.} \end{cases}$$

where $\langle t, v\rangle = \epsilon.N,\ if\ (t \in ObjectsType)\ \text{then}\ \ v = ref.v.$

$$\mathcal{CR}\,[\![N\ D]\!]\ \epsilon \quad = \quad \begin{cases} Error & \text{if } ((\epsilon.N \in \{Error, \bot\}) \vee (t \notin ArrayType) \vee (s' > s)) \\ v & \text{otherwise.} \end{cases}$$

where $\langle t, v\rangle = \epsilon.N,\ t = \langle ty, d, s\rangle,$

$\langle d', s'\rangle = \mathcal{INDT}\,[\![D]\!]\ \epsilon,\ v = nth(v, s'),$

if $(ty \in ObjectType)$ then $v = ref.v.$

$$\mathcal{CR}[\![(N_1)N_2\,.\,N_3]\!]\ \epsilon \;=\; \begin{cases} \mathit{Error} & \text{if } ((\epsilon.N_2 \in \{\mathit{Error}, \bot\}) \vee (t \notin \mathit{ObjectType})) \\ v & \text{otherwise.} \end{cases}$$

where $\langle t, o\rangle = \epsilon.N_2,$ if $o(N_1, N_3) = \bot,$

then if $\mathcal{VT}\ (o\ (\zeta, N_3)) \in \mathit{ObjectsType},$ then $v = ref.o(\zeta, N_3),$

else $v = o(\zeta, N_3).$

else if $\mathcal{VT}\ (o\ (N_1, N_3)) \in \mathit{ObjectsType},$ then $v = ref.o(N_1, N_3),$

else $v = o(N_1, N_3).$

$$\mathcal{CR}[\![N_1\,.\,N_2]\!]\ \epsilon \;=\; \begin{cases} \mathit{Error} & \text{if } ((\epsilon.N_1 \in \{\mathit{Error}, \bot\}) \vee (t \notin \mathit{ObjectType})) \\ v & \text{otherwise.} \end{cases}$$

where $\langle t, o\rangle = \epsilon.N_1,$ if $\mathcal{VT}\ (o\ (\zeta, N_2)) \in \mathit{ObjectsType},$ then $v = ref.o(\zeta, N_2),$

else $v = o(\zeta, N_2).$

$$\mathcal{CR}[\![CR\,.\,(N_1)N_2\,.\,N_3]\!]\ \epsilon \;=\; \begin{cases} \mathit{Error} & \text{if } ((\mathcal{CR}[\![CR\,.\,N_2]\!]\ \epsilon \in \{\mathit{Error}, \bot\}) \vee (t \notin \mathit{ObjectType})) \\ v & \text{otherwise.} \end{cases}$$

where $\langle t, o\rangle = \mathcal{CR}[\![CR\,.\,N_2]\!]\ \epsilon,$

if $o(N_1, N_3) = \bot,$ then if $\mathcal{VT}\ (o\ (\zeta, N_3)) \in \mathit{ObjectsType},$ then $v = ref.o(\zeta, N_3),$

else $v = o(\zeta, N_3).$

else if $\mathcal{VT}\ (o\ (N_1, N_3)) \in \mathit{ObjectsType},$ then $v = ref.o(N_1, N_3),$

else $v = o(N_1, N_3).$

$$\mathcal{CR}[\![CR\,.\,N_1\,.\,N_2]\!]\ \epsilon \;=\; \begin{cases} \mathit{Error} & \text{if } ((\mathcal{CR}[\![CR\,.\,N_1]\!]\ \epsilon \in \{\mathit{Error}, \bot\}) \vee (t \notin \mathit{ObjectType})) \\ v & \text{otherwise.} \end{cases}$$

where $\langle t, o\rangle = \mathcal{CR}[\![CR\,.\,N_1]\!]\ \epsilon,$ if $\mathcal{VT}\ (o\ (\zeta, N_2)) \in \mathit{ObjectsType},$ then $v = ref.o(\zeta, N_2),$

else $v = o(\zeta, N_2).$

$$\mathcal{CR}[\![N.\texttt{Dim[}I\texttt{]}]\!]\ \epsilon \;=\; \begin{cases} \mathit{Error} & \text{if } ((\epsilon.N \in \{\mathit{Error}, \bot\}) \vee (t \notin \mathit{ArrayType}) \vee (I > len(d))) \\ d_I & \text{otherwise.} \end{cases}$$

where $\langle t, v\rangle = \epsilon.N,\ t = \langle ty, d, s\rangle,$

$d_I = \mathrm{I}^{th}$ element of list $d.$

$$\mathcal{CR}[\![N.\texttt{Rank}]\!]\ \epsilon \;=\; \begin{cases} \mathit{Error} & \text{if } ((\epsilon.N \in \{\mathit{Error}, \bot\}) \vee (t \notin \mathit{ArrayType})) \\ r & \text{otherwise.} \end{cases}$$

where $\langle t, v\rangle = \epsilon.N,\ t = \langle ty, d, s\rangle,\ r = len(d).$

$$\mathcal{LT}\left[\!\!\left[\begin{array}{l}\texttt{(let ((}N_1\ CR_1\texttt{)}\\ \qquad\texttt{(}N_2\ CR_2\texttt{)}\\ \qquad\ldots\\ \qquad\texttt{(}N_n\ CR_n\texttt{))}\\ \quad \_LT\_\texttt{)}\end{array}\right]\!\!\right]\ \epsilon \;=\; \begin{cases} \mathit{Error} & \text{if } (\exists\ i :: \mathcal{CR}\,[\![CR_i]\!]\ \epsilon \in \{\bot, \mathit{Error}\}) \\ \langle \mathcal{VT}\,(e)\,, e\rangle & \text{otherwise.}\end{cases}$$

$$\text{where } e = \begin{array}{r} exec(let((N_1\ \mathcal{CR}\,[\![CR_1]\!]\ \epsilon)\ldots \\ (N_n\ \mathcal{CR}\,[\![CR_n]\!]\ \epsilon) \\ LT) \end{array}$$

$$\mathit{append\_fields}(f_1, f_2, N_2) \;=\; \forall \langle m_1, m_2\rangle \in \mathit{dom}.f_2,\ \text{if } \langle m_1, m_2\rangle \in \mathit{dom}.f_1 \text{ then } f_1(N_2, m_2) \mapsto f_2(m_1, m_2),$$
$$\text{else } f_1(m_1, m_2) \mapsto f_2(m_1, m_2).$$

A.2.3 Constraints

$$\mathcal{C}\left[\!\!\left[\begin{array}{r}\texttt{Map } N\ \texttt{((}C_{1,1}, \ldots, C_{1,n_1}\texttt{)}\\ \texttt{(}C_{2,1}, \ldots, C_{2,n_2}\texttt{))}\end{array}\right]\!\!\right]\ \epsilon\ c \;=\; \begin{cases} \mathit{Error} & \text{if } e \\ \mathit{map\_cnstr}(N, D_1, D_2)\ \wedge\ \mathcal{C}\,[\![\mathit{first}(c)]\!]\ \epsilon'\ \mathit{rest}(c) & \text{otherwise.}\end{cases}$$

$$\text{where } D_1 = \bigcup_{i=1}^{n_1} \mathit{sts}(\mathcal{CR}\,[\![C_{1,i}]\!]\ \epsilon),\ D_2 = \bigcup_{i=1}^{n_1} \mathit{sts}(\mathcal{CR}\,[\![C_{2,i}]\!]\ \epsilon),$$
$$D_{i,j} = \mathcal{CR}\,[\![C_{i,j}]\!]\ \epsilon,$$
$$e = ((\exists i, j :: \mathcal{CR}\,[\![C_{i,j}]\!]\ \epsilon \in \{\mathit{Error}, \bot\}) \vee (D_1 \cap D_2 \neq \{\}))$$
$$\epsilon' = \epsilon[N \mapsto \langle (D_{1,i})_{i=1}^{n_1}, (D_{2,j})_{j=1}^{n_2}, (\mathit{dom}.x \mapsto D1, \mathit{img}.x \mapsto D2)\rangle].$$

$$\mathcal{C}\left[\!\!\left[\begin{array}{r}\texttt{Constraint } N\ \texttt{((}L_{1,1,1}, \ldots, L_{1,1,m}\texttt{)}\ldots\\ \texttt{(}L_{1,n_1,1}, \ldots, L_{1,n_1,m}\texttt{)),}\\ \texttt{((}L_{2,1,1}, \ldots, L_{2,1,m}\texttt{)}\ldots\\ \texttt{(}L_{2,n_2,1}, \ldots, L_{2,n_2,m}\texttt{))}\end{array}\right]\!\!\right]\ \epsilon\ c =$$

$$\begin{cases} \mathit{Error} & \text{if } e \\ f\ \wedge\ \mathcal{C}\,[\![\mathit{first}(c)]\!]\ \epsilon\ \mathit{rest}(c) & \text{otherwise.}\end{cases}$$

$$\text{where } \langle (D_{1,i})_{i=1}^{n_1}, (D_{2,j})_{j=1}^{n_2}, x\rangle = \epsilon\,.\,N$$
$$e = ((\epsilon\,.\,N = \mathit{Error}) \vee (\mathcal{VT}\,(\epsilon\,.\,N) \notin \mathit{MapType}) \vee$$
$$\exists i, j, k, c_1, c_2 :: \mathcal{FC}\,(N, \mathit{nth}(D_{1,i}, c_1), L_{1,i,j}, \mathit{nth}(D_{2,k}, c_2)) \neq L_{2,k,j} \vee$$
$$\exists i, j, c_1 :: \mathcal{VT}\,(\mathcal{CR}\,[\![\mathit{nth}(D_{1,i}, c_1).L_{1,i,j}]\!]\ \epsilon) \notin \mathbb{Z} \vee$$
$$\exists j, k, c_2 :: \mathcal{VT}\,(\mathcal{CR}\,[\![\mathit{nth}(D_{2,k}, c_2).L_{2,k,j}]\!]\ \epsilon) \notin \mathbb{Z})$$
$$f = \mathit{fld\_cnstr}(\epsilon\,.\,N, ((L_{1,i,j})_{j=1}^{m})_{i=1}^{n_1}, ((L_{2,i,j})_{j=1}^{m})_{i=1}^{n_2})$$

$$\mathcal{C}\,[\![\texttt{For\_all}\ N_1\ \texttt{in}\ N_2\ e]\!]\ \epsilon\ c \quad = \quad \begin{cases} \mathit{Error} & \text{if } ((\mathcal{VT}\,(\mathcal{CR}\,[\![N_2]\!]\ \epsilon) \notin \mathbb{N} \cup \mathit{ArrayType}) \wedge N_2 \notin \mathbb{N}) \\ \mathcal{C}\,[\![\texttt{For}\ N_1\texttt{at}\ 0\ \texttt{in}\ N_2\ e]\!]\ \epsilon\ c & \text{if } ((\mathcal{VT}\,(\mathcal{CR}\,[\![N_2]\!]\ \epsilon) = \mathbb{N}) \vee N_2 \in \mathbb{N}) \\ \mathcal{C}\,[\![\texttt{For}\ N_1\texttt{is}\ \mathit{nth}(N_2, 0)\ \texttt{in}\ N_2\ e]\!]\ \epsilon\ c & \text{if } (\mathcal{VT}\,(\mathcal{CR}\,[\![N_2]\!]\ \epsilon) \in \mathit{ArrayType}) \end{cases}$$

$$\mathcal{C}\,[\![\texttt{For}\ N_1\texttt{at}\ I\ \texttt{in}\ N_2\ e]\!]\ \epsilon\ c \quad = \quad \begin{cases} \mathcal{C}\,[\![\mathit{first}(c)]\!]\ \epsilon\ \mathit{rest}(c) & \text{if } ((I \geq \mathcal{CR}\,[\![N_2]\!]\ \epsilon)\ \vee\ (I \geq N_2)) \\ \mathcal{C}\,[\![e]\!]\ \epsilon[N_1 \mapsto I]\ ((\texttt{For}\ N_1\texttt{at}\ I + 1\ \texttt{in}\ N_2\ e); c) & \text{otherwise.} \end{cases}$$

$$\mathcal{C}\,[\![\texttt{For}\ N_1\texttt{is}\ \mathit{nth}(N_2, I)\ \texttt{in}\ N_2\ e]\!]\ \epsilon\ c \quad = \quad \begin{cases} \mathcal{C}\,[\![\mathit{first}(c)]\!]\ \epsilon\ \mathit{rest}(c) & \text{if } (I \geq s, \langle t, d, s\rangle = \mathcal{VT}\,(\mathcal{CR}\,[\![N_2]\!]\ \epsilon)) \\ \mathcal{C}\,[\![e]\!]\ \epsilon'\ ((\texttt{For}\ N_1\texttt{is}\ \mathit{nth}(N_2, I+1)\ \texttt{in}\ N_2\ e); c) & \text{otherwise.} \end{cases}$$
where $\epsilon' = \epsilon[N_1 \mapsto \mathit{nth}(N_2, I)]$.

$$\mathcal{C}\,[\![F_1\ \texttt{and}\ F_2]\!]\ \epsilon\ c \quad = \quad \mathcal{C}\,[\![F_1]\!]\ \epsilon\ (F_2); c$$

$$\mathcal{C}\,[\![B]\!]\ \epsilon\ c \quad = \quad \mathit{apply\ and}\ (\mathcal{B}\,[\![B]\!]\ \epsilon,\ \mathcal{C}\,[\![\mathit{first}(c)]\!]\ \epsilon\ \mathit{rest}(c)).$$

$$\mathcal{C}\,[\![CR\ \texttt{=}\ T]\!]\ \epsilon\ c \quad = \quad \begin{cases} \mathit{Error} & \text{if } ((\mathcal{CR}\,[\![CR]\!]\ \epsilon \neq \bot) \vee (\mathcal{VT}\,(\mathcal{T}\,[\![T]\!]\ \epsilon) \neq \mathcal{VT}\,(\mathcal{CR}\,[\![CR]\!]\ \epsilon))) \\ \mathcal{C}\,[\![\mathit{first}(c)]\!]\ \mathit{set}(CR, (\mathcal{T}\,[\![T]\!]\ \epsilon), \epsilon)\ c & \text{otherwise.} \end{cases}$$

$$\mathcal{C}\,[\![T_1\ RT\ T_2]\!]\ \epsilon\ c \quad = \quad \mathit{apply}\ \mathcal{OP}\,[\![RT]\!]\,(\mathcal{T}\,[\![T_1]\!]\ \epsilon, \mathcal{T}\,[\![T_2]\!]\ \epsilon)\ \ \wedge\ \mathcal{C}\,[\![\mathit{first}(c)]\!]\ \epsilon\ \mathit{rest}(c).$$

$$\mathcal{T}\,[\![\texttt{Sum}\ N_1\ \texttt{in}\ N_2\ T]\!]\ \epsilon \quad = \quad \begin{cases} \mathit{Error} & \text{if } ((\mathcal{VT}\,(\mathcal{CR}\,[\![N_2]\!]\ \epsilon) \notin \mathbb{N} \cup \mathit{ArrayType}) \wedge N_2 \notin \mathbb{N}) \\ \mathit{apply}\ \mathcal{OP}\,[\![\texttt{+}]\!]\,(T_i)_{i=0}^{N} & \text{if } ((\mathcal{VT}\,(\mathcal{CR}\,[\![N_2]\!]\ \epsilon) = \mathbb{N}) \vee N_2 \in \mathbb{N}) \\ \mathit{apply}\ \mathcal{OP}\,[\![\texttt{+}]\!]\,(T_j)_{j=0}^{N} & \text{if } (\mathcal{VT}\,(\mathcal{CR}\,[\![N_2]\!]\ \epsilon) \in \mathit{ArrayType}) \end{cases}$$
where $N = N_2$ if $N_2 \in \mathbb{N}$

$N = s, \langle t, d, s\rangle = \mathcal{VT}\,(\mathcal{CR}\,[\![N_2]\!]\ \epsilon)$ if $(\mathcal{VT}\,(\mathcal{CR}\,[\![N_2]\!]\ \epsilon) = \mathit{ArrayType})$

$N = \mathcal{VT}\,(\mathcal{CR}\,[\![N_2]\!]\ \epsilon)$ if $(\mathcal{VT}\,(\mathcal{CR}\,[\![N_2]\!]\ \epsilon) = \mathbb{N})$

$\forall\ T_i = \mathcal{T}\,[\![T]\!]\ \epsilon[N_1 \mapsto i]$

$\forall\ T_j = \mathcal{T}\,[\![T]\!]\ \epsilon[N_1 \mapsto \mathit{nth}(N_2, j)]$.

$$\mathcal{T}\,[\![(\mathit{op}\ T_1\ \ldots\ T_n)]\!]\ \epsilon \quad = \quad \mathit{apply}\ \mathcal{OP}\,[\![\mathit{op}]\!]\,(\mathcal{T}\,[\![T_i]\!]\ \epsilon)_{i=1}^{n}.$$

$$\mathcal{T}\,[\![CR]\!]\ \epsilon \quad = \quad \begin{cases} \mathit{Error} & \text{if } (v = \mathit{Error} \vee (\mathcal{VT}\,(v) \notin \mathbb{Z} \cup \mathit{Bool}) \vee (v = \bot\ \ \wedge\ (\mathcal{VT}\,(v) \neq \mathit{Bool}))) \\ \mathit{var}(CR) & \text{if } (v = \bot\ \wedge\ (\mathcal{VT}\,(v) = \mathit{Bool}) \\ v & \text{otherwise.} \end{cases}$$
where $v = \mathcal{CR}\,[\![CR]\!]\ \epsilon$

$$\mathcal{T}\,[\![MR]\!]\ \epsilon \quad = \quad \begin{cases} \mathit{Error} & \text{if } m = \mathit{Error} \\ \mathit{var}(CR) & \text{if } m = \bot \\ 1 & \text{if } m = \texttt{True} \\ 0 & \text{if } m = \texttt{False} \end{cases} \quad \text{where } m = \mathcal{MR}\,[\![MR]\!]\ \epsilon$$

$$\mathcal{T}\,[\![I]\!]\ \epsilon \quad = \quad \begin{cases} \mathit{Error} & \text{if } I \notin \mathbb{Z} \\ I & \text{otherwise.} \end{cases}$$

$$\mathcal{T}\,[\![\texttt{(let}\ \ lt\texttt{)}]\!]\ \epsilon \quad = \quad \begin{cases} \mathit{Error} & \text{if } t \neq \mathbb{Z} \\ \langle t, v\rangle = \mathcal{LT}\,[\![\texttt{(let}\ \ lt\texttt{)}]\!]\ \epsilon & \text{otherwise.} \end{cases}$$

$$\mathcal{B}\,[\![BE_1\ op\ BE_2]\!]\ \epsilon \quad = \quad \mathit{apply}\ \mathcal{OP}\,[\![op]\!]\,(\mathcal{B}\,[\![BE_1]\!]\ \epsilon, \mathcal{B}\,[\![BE_2]\!]\ \epsilon).$$

$$\mathcal{B}\,[\![\texttt{Not}\ BE]\!]\ \epsilon \quad = \quad \mathit{apply}\ \mathcal{OP}\,[\![\texttt{Not}]\!]\,(\mathcal{B}\,[\![BE]\!]\ \epsilon).$$

$$\mathcal{B}\,[\![CR]\!]\ \epsilon \quad = \quad \begin{cases} \mathit{Error} & \text{if } (v = \mathit{Error} \vee (\mathcal{VT}\,(v) \notin \mathbb{Z} \cup \mathit{Bool}) \vee (v = \bot \ \wedge\ (\mathcal{VT}\,(v) = \mathbb{Z}))) \\ \mathit{var}(CR) & \text{if } ((v = \bot)\ \wedge\ (\mathcal{VT}\,(v) = \mathit{Bool})) \\ \mathit{True} & \text{if } v \\ \mathit{False} & \text{otherwise.} \end{cases}$$
where $v = \mathcal{CR}\,[\![CR]\!]\ \epsilon$

$$\mathcal{B}\,[\![MR]\!]\ \epsilon \quad = \quad \begin{cases} \mathit{Error} & \text{if } m = \mathit{Error} \\ \mathit{var}(MR) & \text{if } m = \bot \\ m & \text{otherwise.} \end{cases} \quad \text{where } m = \mathcal{MR}\,[\![MR]\!]\ \epsilon$$

$$\mathcal{B}\,[\![\texttt{(let}\ \ lt\texttt{)}]\!]\ \epsilon \quad = \quad \begin{cases} \mathit{Error} & \text{if } t \neq \mathit{Bool} \\ \langle t, v\rangle = \mathcal{LT}\,[\![\texttt{(let}\ \ lt\texttt{)}]\!]\ \epsilon & \text{otherwise.} \end{cases}$$

A.2.4 Objective

$$\mathcal{O}\,[\![\mathit{Maximize}\ T_1\ RT\ T_2]\!]\ \epsilon \quad = \quad \mathit{maximize}\ (\mathit{apply}\ \mathcal{OP}\,[\![RT]\!]\,(\mathcal{T}\,[\![T_1]\!]\ \epsilon, \mathcal{T}\,[\![T_2]\!]\ \epsilon))$$

$$\mathcal{O}\,[\![\mathit{Minimize}\ T_1\ RT\ T_2]\!]\ \epsilon \quad = \quad \mathit{minimize}\ (\mathit{apply}\ \mathcal{OP}\,[\![RT]\!]\,(\mathcal{T}\,[\![T_1]\!]\ \epsilon, \mathcal{T}\,[\![T_2]\!]\ \epsilon))$$

REFERENCES

[1] Aloul, F., Ramani, A., Markov, I., and Sakallah, K., "Generic ILP versus specialized 0-1 ILP: an update," in *International Conference on Computer Aided Design (ICCAD)*, 2002.

[2] Aloul, F., Ramani, A., Markov, I., and Sakallah, K., "PBS: A backtrack search pseudo-Boolean solver," in *Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, 2002.

[3] Barth, P., "A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization," Research Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, January 1995.

[4] Bass, L., Clements, P., and Kazman, R., *Software Architecture in Practice*. Addison-Wesley, 1998.

[5] Batory, D. and Geraci., B. J., "Composition validation and subjectivity in GenVoca generators," *IEEE Transactions on Software Engineering (IEEE TSE)*, pp. 67–82, 1997.

[6] Batory, D. and O'Malley, S., "The design and implementation of hierarchical software systems with reusable components," *ACM Transaction Software Engineering Methodology*, vol. 1, no. 4, pp. 355–398, 1992.

[7] Batory, D., Singhal, V., Thomas, J., Dasari, S., Geraci, B., and Sirkin, M., "The GenVoca model of software-system generators," *Software, IEEE*, vol. 11, pp. 89–94, Sep 1994.

[8] Bertolino, A. and Mirandola, R., "Modeling and analysis of non-functional properties in component-based systems," in *TACoS 2003: Proc. International Workshop on Test and Analysis of Component Based Systems*, vol. 82 of *Electronic Notes in Theoretical Computer Science*, April 2003.

[9] Cao, F., Bryant, B. R., Burt, C. C., Raje, R. R., Olson, A. M., and Auguston, M., "A component assembly approach based on aspect-oriented generative domain modeling," *Electronic Notes in Theoretical Computer Science*, vol. 114, pp. 119–136, January 2005.

[10] Chai, D. and Kuehlmann, A., "A fast pseudo-Boolean constraint solver," in *Proc. of the 40th Design Automation Conference (DAC 2003)*, 2003.

[11] Cheesman, J. and Daniels, J., *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2000.

[12] Committee Arinc 651, "Arinc report 651, draft 9," Technical Report 91-207/SAI-435, Airlines Electronic Engineering Committee, September 1991.

[13] Crnkovic, I., "Component-based software engineering - new challenges in software development," *Software Focus*, December 2001.

[14] Crnkovic, I. and Larsson, M., *Building Reliable Component-Based Software Systems*. Artech House publisher, 2002.

[15] Crnkovic, I., Schmidt, H., Stafford, J., and Wallnau, K., eds., *4th Workshop on Component-Based Software Engineering*, (Toronto, Canada), ACM, IEEE, May 2001. 23nd International Conference on Software Engineering (ICSE2001).

[16] Crnkovic, I., Schmidt, H., Stafford, J., and Wallnau, K., "Automated component-based software engineering," *Journal of Systems and Software*, vol. 74, January 2005.

[17] Davis, M., Logemann, G., and Loveland, D., "A machine program for theorem proving," *Communications of the ACM*, vol. 5, pp. 394–397, 1962.

[18] Davis, M. and Putnam, H., "A computing procedure for quantification theory," *Journal of ACM*, vol. 7, pp. 201–215, 1960.

[19] Eén, N. and Sörensson, N., "An extensible SAT-solver," in *Proc. SAT 03*, 2003. `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/-MiniSat.html`, Date accessed: May 1, 2006.

[20] Eén, N. and Sörensson, N., "Translating pseudo-Boolean constraints into SAT," in *Journal on Satisfiability Boolean Modeling and Computation*, vol. 2, pp. 1–26, 2006.

[21] Feldmann, R., Haubelt, C., Monien, B., and Teich, J., "Fault tolerance analysis of distributed reconfigurable systems using SAT-based techniques," in *Field-Programmable Logic and Applications* (Cheung, P. Y. K., Constantinides, G. A., and de Sousa, J. T., eds.), (Berlin, Heidelberg), pp. 478–487, Springer, September 2003.

[22] Fredriksson, J., Åkerholm, M., and Sandström, K., "Calculating resource tradeoffs when mapping component services to real-time tasks," in *Fourth Conference on Software Engineering Research and Practice in Sweden Linköping, Sweden*, October 2004.

[23] Garlan, D., "Architectural mismatch or why it's hard to build systems out of existing parts," in *International Conference on Software Engineering*, 1995.

[24] Hammer, D. K. and Chaudron, M., "Towards component-based architecting for resource constraint systems," in *Proc. 4rth International Software Architecture Workshop (ISAW-4)*, June 2000.

[25] Heineman, G. T. and Councill, W. T., *Component Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[26] ILOG CPLEX, "`http://www.ilog.com/products/cplex/`." Date accessed: May 1, 2006.

[27] Inverardi, P. and Tivoli, M., "Software architecture for correct components assembly," *Lecture Notes in Computer Science*, vol. 2804, Nov 2003.

[28] Jackson, P. and Sheridan, D., "Clause form conversions for Boolean circuits," in *Post-conference proceedings of SAT 2004 (7th International Conference on Theory and Applications of Satisfiability Testing)*, vol. 3542 of *LNCS*, pp. 183–198, Springer-Verlag, 2004.

[29] Karmarkar, N., "A new polynomial-time algorithm for linear programming," in *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, (New York, NY, USA), pp. 302–311, ACM Press, 1984.

[30] Lacour, S., Perez, C., and Priol, T., "A software architecture for automatic deployment of CORBA components using grid technologies," in *Proceedings of the 1st Franco-phone Conference On Software Deployment and (Re)Configuration (DECOR 2004)*, Oct. 2004.

[31] Lee, E. K. and Mitchell, J. E., "Branch-and-bound methods for integer programming," *Encyclopedia of Optimization*, August 2001.

[32] Manolios, P., Subramanian, G., and Vroon, D., "Automating component-based system assembly." Manuscript submitted for publication, 2006.

[33] Manquinho, V. M. and Roussel, O., "The first evaluation of pseudo-Boolean solvers (pb'05)," in *Journal on Satisfiability Boolean Modeling and Computation*, vol. 2, pp. 103–143, 2006.

[34] Martin, F. and Fraboul, C., "Modeling and simulation of integrated modular avionics," in *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing, 1998. PDP '98.*, pp. 102 – 110, 1998.

[35] Medvidovic, N. and Taylor, R. N., "A classification and comparison framework for software architecture description languages," *IEEE Trans. Software Eng.*, vol. 26, no. 1, pp. 70–93, 2000.

[36] Microsoft Corporation, "Microsoft .NET platform," 2006. `http://www.microsoft.com/net/`, Date accessed: May 1, 2006.

[37] Mikic-Rakic, M., Malek, S., Beckman, N., and Medvidovic, N., "A tailorable environment for assessing the quality of deployment architectures in highly distributed settings.," in *Component Deployment, Second International Working Conference, CD 2004*, pp. 1–17, 2004.

[38] Mitchell, J. E., "Branch-and-cut algorithms for integer programming," *Encyclopedia of Optimization*, August 2001.

[39] Mitchell, J. E., "Cutting plane algorithms for integer programming," *Encyclopedia of Optimization*, August 2001.

[40] Möller, A., Åkerholm, M., Fredriksson, J., and Nolin, M., "Evaluation of component technologies with respect to industrial requirements," in *Euromicro Conference, Component-Based Software Engineering Track*, (Rennes, France), August 2004.

[41] Object Management Group, Inc., "MinimumCORBA 1.0," August 1998. Joint Revised Submission, OMG Document orbos/98-08-04 ed.

[42] Object Management Group, Inc., "CORBA component model 3.0," 2006. http://www.omg.org/technology/documents/formal/components.htm, Date accessed: May 1, 2006.

[43] Object Management Group (OMG), "Response to the UML 2.0 OCL RfP Revised Submission, Version 1.6," 2003. http://www.omg.org/docs/ad/03-01-07.pdf, Date accessed: May 1, 2006.

[44] Oreizy, P., Medvidovic, N., Taylor, R., and Rosenblum, D., "Software architecture and component technologies: Bridging the gap," in *Digest of the OMG-DARPA-MCC Workshop on Compositional Software Architectures*, January 1998.

[45] Oreizy, P., Rosenblum, D. S., and Taylor, R. N., "On the role of connectors in modeling and implementing software architectures," Tech. Rep. ICS-TR-98-04, Department of Information and Computer Science, University of California, Irvine, 1998.

[46] Papadimitriou, C. H., "On the complexity of integer programming," *ACM Journal*, vol. 28, no. 4, pp. 765–768, 1981.

[47] Perry, D. E. and Wolf, A. L., "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[48] Prisaznuk, P. J., "Integrated modular avionics," in *Proceedings of the IEEE 1992 National Aerospace and Electronics Conference (NAECON 1992)*, vol. 1, pp. 39 – 45, 1992.

[49] Rational Partners, Object Management Group, "UML Notation Guide," Sept. 1997. http://www.omg.org/docs/ad/97-08-04.pdf, Date accessed: May 1, 2006.

[50] Rational Partners, Object Management Group, "UML Semantics," Sept. 1997. http://www.omg.org/docs/ad/97-08-04.pdf, Date accessed: May 1, 2006.

[51] Sánchez-Puebla, M. A. and Carretero, J., "A new approach for distributed computing in avionics systems," in *ISICT '03: Proceedings of the 1st international symposium on Information and communication technologies*, pp. 579–584, Trinity College Dublin, 2003.

[52] Sandström, K., Fredriksson, J., and Åkerholm, M., "Introducing a component technology for safety critical embedded realtime systems," in *International Symposium on Component-based Software Engineering (CBSE7)*, (Edinburgh, Scotland), Springer Verlag, May 2004.

[53] Schmidt, D. C. and Kuhns, F., "An overview of the real-time CORBA specification," *IEEE Computer Magazine, Special Issue on Object-oriented Real-time Computing*, vol. 33, no. 6, pp. 56–63, 2000.

[54] Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., and Zelesnik, G., "Abstractions for software architecture and tools to support them," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 314–335, 1995.

[55] Sheini, H. M. and Sakallah, K. A., "Pueblo: A modern pseudo-Boolean SAT solver," in *Proceedings of the Design, Automation and Test in Europe (DATE'05)*, vol. 2, pp. 684 – 685, 2005.

[56] Sheini, H. M. and Sakallah, K. A., "A SAT-based decision procedure for mixed logical/integer linear problems.," in *CPAIOR: International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pp. 320–335, 2005.

[57] Sun Developer Network (SDN), "J2EE Enterprise Javabeans Technology," 2005. `http://java.sun.com/products/ejb/`, Date accessed: May 1, 2006.

[58] Szyperski, C., Gruntz, D., and Murer, S., *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, second ed., 2002.

[59] Tindell, Burns, and Wellings, "Allocating hard real-time tasks: An NP-hard problem made easy," *RTSYSTS: Real-Time Systems*, vol. 4, 1992.

[60] van der Hoek, A., Heimbigner, D., and Wolf, A., "Software architecture, configuration management, and configurable distributed systems: A ménage a trois," Technical Report CU-CS-849-98, Department of Computer Science, University of Colorado at Boulder, 1998.

[61] van Ommering, R., van der Linden, F., Kramer, J., and Magee, J., "The Koala component model for consumer electronics software," *Computer*, vol. 33, no. 3, pp. 78–85, 2000.

[62] Wallnau, K., Hissam, S., and Seacord, R., *Building Systems from Commercial Components*. SEI Series in Software Engineering, Addison-Wesley, 2001.

[63] Wallnau, K., Stafford, J., Hissam, S., and Klein, M., "On the relationship of software architecture to software component technology," in *Proceedings of the 6th ECOOP Workshop on Component-Oriented Programming*, 2001.

[64] Wallnau, K. C., "Volume III: A Technology for Predictable Assembly from Certifiable Components (PACC)," Tech. Rep. CMU/SEI-2003-TR-009, Carnegie Mellon University, April 2003.

[65] Wang, Gill, C., Schmidt, D. C., and Subramonian, V., "Configuring real-time aspects in component middleware," in *Proceedings of the Conference on Distributed Objects and Applications (DOA 2004)*, 2004.

[66] Warrilow, R., "The avionics platform," 2004. `www.smiths-aerospace.com/Press-/TechPapers/`, Date accessed: May 1, 2006.

[67] Weyuker, E. J., "Testing component-based software: A cautionary tale," *IEEE Software*, vol. 15, no. 5, pp. 54–59, 1998.

[68] Wuyts, R., Ducasse, S., and Nierstrasz, O., "A data-centric approach to composing embedded, real-time software components," *Journal of Systems and Software*, vol. 74, pp. 25–34, January 2005.

[69] Zhang, L., Madigan, C., Moskewicz, M., and Malik, S., "Efficient conflict driven learning in a Boolean satisfiability solver," in *Proceedings of ICCAD 2001*, Nov. 2001.

[70] Zhang, L. and Malik, S., "The quest for efficient Boolean satisfiability solvers," in *18th International Conference on Automated Deduction, CADE'02*, pp. 295–313, 2002.